

$\langle Version \rangle \equiv$   
2.2.0\_alpha-1  
 $\langle Date \rangle \equiv$   
Sep 25 2013

# WHIZARD<sup>1</sup>

Wolfgang Kilian,<sup>2</sup> Thorsten Ohl,<sup>3</sup> Jürgen Reuter<sup>4</sup>

Version 2.2.0  $\alpha$ -1, Sep 25 2013

with contributions from: Fabian Bach, Felix Braam, David Gordo  
Gomez, Sebastian Schmidt, Marco Sekulla, Christian Speckner,  
Daniel Wiesler

<sup>1</sup>The original meaning of the acronym is *W, Higgs, Z, And Respective Decays*. The current program is much more than that, however.

<sup>2</sup>e-mail: [kilian@physik.uni-siegen.de](mailto:kilian@physik.uni-siegen.de)

<sup>3</sup>e-mail: [ohl@physik.uni-wuerzburg.de](mailto:ohl@physik.uni-wuerzburg.de)

<sup>4</sup>e-mail: [juergen.reuter@desy.de](mailto:juergen.reuter@desy.de)

### **Abstract**

WHIZARD is an application of the VAMP algorithm: Adaptive multi-channel integration and event generation. The bare VAMP library is augmented by modules for Lorentz algebra, particles, phase space, etc., such that physical processes with arbitrary complex final states [well, in principle. . .] can be integrated and *unweighted* events be generated.

# Contents

<b>1</b>	<b>Changes</b>	<b>17</b>
<b>2</b>	<b>Tools</b>	<b>18</b>
2.1	Preliminaries . . . . .	18
2.2	Parallelization . . . . .	20
2.3	File utilities . . . . .	20
2.3.1	Finding an I/O unit . . . . .	20
2.3.2	Deleting a file . . . . .	21
2.3.3	String auxiliary functions . . . . .	22
2.3.4	Formatting numbers . . . . .	24
2.4	Unit tests . . . . .	25
2.4.1	Parameters . . . . .	25
2.4.2	Type for storing test results . . . . .	26
2.4.3	Wrapup . . . . .	28
2.4.4	Tool for Unit Tests . . . . .	28
2.5	Operating-system interface . . . . .	30
2.5.1	Path variables . . . . .	30
2.5.2	System dependencies . . . . .	31
2.5.3	Dynamic linking . . . . .	36
2.5.4	Predicates . . . . .	40
2.5.5	Shell access . . . . .	40
2.5.6	Querying for a directory . . . . .	41
2.5.7	Fortran compiler and linker . . . . .	41
2.5.8	Controlling OpenMP . . . . .	44
2.5.9	Test . . . . .	45
2.6	CPU timing . . . . .	47
2.7	Accessing the system clock . . . . .	50
2.8	Hashtables . . . . .	57
2.8.1	The hash function . . . . .	57
2.8.2	The hash table . . . . .	58
2.8.3	Hashtable insertion . . . . .	60
2.8.4	Hashtable lookup . . . . .	61
2.9	Message and signal handling . . . . .	62
2.9.1	Logfile . . . . .	72
2.9.2	Checking values . . . . .	73
2.9.3	Signal handling . . . . .	76
2.10	C wrapper for sigaction . . . . .	78
2.11	C wrapper for printf . . . . .	80

2.12	Interface for formatted I/O . . . . .	82
2.12.1	Parsing a C format string . . . . .	82
2.12.2	API . . . . .	89
2.12.3	Test . . . . .	90
2.13	Bytes and such . . . . .	91
2.13.1	8-bit words: bytes . . . . .	91
2.13.2	32-bit words . . . . .	93
2.13.3	Operations on 32-bit words . . . . .	96
2.13.4	64-bit words . . . . .	98
2.14	MD5 Checksums . . . . .	101
2.14.1	Blocks . . . . .	101
2.14.2	Messages . . . . .	103
2.14.3	Message I/O . . . . .	105
2.14.4	Auxiliary functions . . . . .	107
2.14.5	Auxiliary stuff . . . . .	107
2.14.6	MD5 algorithm . . . . .	108
2.14.7	User interface . . . . .	111
2.15	Permutations . . . . .	112
2.15.1	Permutations . . . . .	113
2.15.2	Operations on binary codes . . . . .	118
2.16	Sorting . . . . .	119
2.16.1	Implementation . . . . .	119
2.16.2	Concatenating arrays . . . . .	121
2.16.3	Test . . . . .	122
<b>3</b>	<b>Text handling</b>	<b>124</b>
3.1	Internal files . . . . .	125
3.1.1	iostat codes . . . . .	125
3.1.2	The line type . . . . .	125
3.1.3	The ifile type . . . . .	126
3.1.4	I/O on ifiles . . . . .	127
3.1.5	Ifile tools . . . . .	130
3.1.6	Line pointers . . . . .	130
3.1.7	Access lines via pointers . . . . .	131
3.2	Lexer . . . . .	133
3.2.1	Input streams . . . . .	134
3.2.2	Keyword list . . . . .	137
3.2.3	Lexeme templates . . . . .	139
3.2.4	The lexer setup . . . . .	144
3.2.5	The lexeme type . . . . .	147
3.2.6	The lexer object . . . . .	150
3.2.7	The lexer routine . . . . .	151
3.2.8	Diagnostics . . . . .	154
3.3	Syntax rules . . . . .	156
3.3.1	Syntax rules . . . . .	156
3.3.2	I/O . . . . .	159
3.3.3	Completing syntax rules . . . . .	161
3.3.4	Accessing rules . . . . .	163
3.3.5	Syntax tables . . . . .	165
3.3.6	Accessing the syntax table . . . . .	171

3.3.7	I/O . . . . .	172
3.4	The parser . . . . .	174
3.4.1	The token type . . . . .	174
3.4.2	Retrieve token contents . . . . .	178
3.4.3	The parse tree: nodes . . . . .	180
3.4.4	Filling nodes . . . . .	182
3.4.5	Accessing nodes . . . . .	184
3.4.6	The parse tree . . . . .	188
3.4.7	Access the parser . . . . .	194
3.4.8	Tools . . . . .	194
3.4.9	Applications . . . . .	195
3.4.10	Test the parser . . . . .	196
<b>4</b>	<b>Physics library</b>	<b>199</b>
4.1	C-compatible Particle Type . . . . .	200
4.2	Lorentz algebra . . . . .	202
4.2.1	Three-vectors . . . . .	202
4.2.2	Four-vectors . . . . .	208
4.2.3	Conversions . . . . .	214
4.2.4	Angles . . . . .	216
4.2.5	More kinematical functions (some redundant) . . . . .	222
4.2.6	Lorentz transformations . . . . .	226
4.2.7	Functions of Lorentz transformations . . . . .	227
4.2.8	Invariants . . . . .	227
4.2.9	Boosts . . . . .	228
4.2.10	Rotations . . . . .	230
4.2.11	Composite Lorentz transformations . . . . .	232
4.2.12	Applying Lorentz transformations . . . . .	233
4.2.13	Special Lorentz transformations . . . . .	233
4.2.14	Special functions . . . . .	236
4.3	Special Physics functions . . . . .	238
4.3.1	Running $\alpha_s$ . . . . .	238
4.3.2	Catani-Seymour Parameters . . . . .	241
4.3.3	Mathematical Functions . . . . .	242
4.3.4	Loop Integrals . . . . .	243
4.3.5	More on $\alpha_s$ . . . . .	245
4.3.6	Functions for Catani-Seymour dipoles . . . . .	246
4.3.7	Distributions for integrated dipoles and such . . . . .	248
4.4	QCD Coupling . . . . .	256
4.4.1	Coupling: Abstract Data Type . . . . .	257
4.4.2	Fixed Coupling . . . . .	258
4.4.3	Running Coupling . . . . .	259
4.4.4	Running Coupling, determined by $\Lambda_{\text{QCD}}$ . . . . .	260
4.4.5	Wrapper type . . . . .	260
4.4.6	Unit tests . . . . .	261

<b>5</b>	<b>Physics Analysis</b>	<b>265</b>
5.1	Analysis tools	266
5.1.1	Output formats	266
5.1.2	Graph options	267
5.1.3	Drawing options	271
5.1.4	Observables	275
5.1.5	Output	278
5.1.6	Histograms	280
5.1.7	Plots	290
5.1.8	Graphs	296
5.1.9	Analysis objects	302
5.1.10	Analysis object iterator	308
5.1.11	Analysis store	311
5.1.12	L <sup>A</sup> T <sub>E</sub> X driver file	313
5.1.13	API	315
5.1.14	Test	323
5.2	PDG arrays	325
5.2.1	Type definition	325
5.2.2	Parameters	326
5.2.3	Basic operations	326
5.2.4	Matching	327
5.3	subevents	329
5.3.1	Particles	329
5.3.2	C-compatible particle type	337
5.3.3	subevents	337
5.3.4	Eliminate numerical noise	349
5.4	User Code Interface	349
5.4.1	User Code Management	350
5.4.2	Libmanager interface	353
5.4.3	Interfaces for user-defined functions	354
5.4.4	Interfaces for user-defined interactions	355
5.5	Variables	357
5.5.1	Variable list entries	358
5.5.2	Setting values	373
5.5.3	Copies and pointer variables	376
5.5.4	Variable lists	379
5.5.5	Tools	387
5.5.6	Process-specific variables	395
5.5.7	Observable initialization	396
5.5.8	Observables	400
5.5.9	Event variables	405
5.5.10	API for variable lists	408
5.5.11	Linking model variables	412
5.6	Expressions	419
5.6.1	Tree nodes	419
5.6.2	Operation types	436
5.6.3	Specific operators	440
5.6.4	Compiling the parse tree	465
5.6.5	Auxiliary functions for the compiler	526
5.6.6	Evaluation	527

5.6.7	Evaluation syntax . . . . .	534
5.6.8	Set up appropriate parse trees . . . . .	543
5.6.9	The evaluation tree . . . . .	545
5.6.10	Direct evaluation . . . . .	553
5.6.11	Test . . . . .	558
<b>6</b>	<b>Physics Models</b>	<b>562</b>
6.1	Model module . . . . .	562
6.1.1	Physics Parameters . . . . .	563
6.1.2	Particle codes . . . . .	565
6.1.3	Spin codes . . . . .	566
6.1.4	Particle data . . . . .	566
6.1.5	Vertex data . . . . .	577
6.1.6	Vertex lookup table . . . . .	578
6.1.7	Model data . . . . .	582
6.1.8	Reading models from file . . . . .	594
6.1.9	Model list . . . . .	603
6.1.10	Test . . . . .	606
<b>7</b>	<b>Quantum Numbers</b>	<b>608</b>
7.1	Helicities . . . . .	609
7.1.1	Helicity types . . . . .	609
7.1.2	Predicates . . . . .	612
7.1.3	Accessing contents . . . . .	613
7.1.4	Comparisons . . . . .	613
7.1.5	Tools . . . . .	615
7.2	Colors . . . . .	616
7.2.1	The color type . . . . .	616
7.2.2	Predicates . . . . .	621
7.2.3	Accessing contents . . . . .	623
7.2.4	Comparisons . . . . .	624
7.2.5	Tools . . . . .	626
7.2.6	Color counting test . . . . .	635
7.2.7	The Madgraph color model . . . . .	636
7.3	Flavors: Particle properties . . . . .	640
7.3.1	The flavor type . . . . .	640
7.4	Quantum numbers . . . . .	655
7.4.1	The quantum number type . . . . .	655
7.4.2	I/O . . . . .	658
7.4.3	Accessing contents . . . . .	660
7.4.4	Predicates . . . . .	662
7.4.5	Comparisons . . . . .	663
7.4.6	Operations . . . . .	666
7.4.7	The quantum number mask . . . . .	669
7.4.8	Setting mask components . . . . .	670
7.4.9	Mask predicates . . . . .	672
7.4.10	Operators . . . . .	672
7.4.11	Mask comparisons . . . . .	672
7.4.12	Apply a mask . . . . .	673
7.5	State matrices . . . . .	676



7.5.1	Nodes of the quantum state trie . . . . .	676
7.5.2	State matrix . . . . .	681
7.5.3	State iterators . . . . .	690
7.5.4	Operations on quantum states . . . . .	697
7.5.5	Factorization . . . . .	703
7.5.6	Test . . . . .	706
7.6	Interactions . . . . .	709
7.6.1	External interaction links . . . . .	710
7.6.2	Internal relations . . . . .	711
7.6.3	The interaction type . . . . .	714
7.6.4	Methods inherited from the state matrix member . . . . .	720
7.6.5	Accessing contents . . . . .	725
7.6.6	Modifying contents . . . . .	732
7.6.7	Handling Linked interactions . . . . .	734
7.6.8	Recovering connections . . . . .	740
7.6.9	Test . . . . .	741
7.7	Matrix element evaluation . . . . .	743
7.7.1	Array of pairings . . . . .	744
7.7.2	The evaluator type . . . . .	744
7.7.3	Auxiliary structures for evaluator creation . . . . .	747
7.7.4	Creating an evaluator: Matrix multiplication . . . . .	756
7.7.5	Creating an evaluator: square . . . . .	767
7.7.6	Accessing contents . . . . .	782
7.7.7	Inherited procedures . . . . .	782
7.7.8	Deleting the evaluator . . . . .	787
7.7.9	Creating an evaluator: identity . . . . .	787
7.7.10	Creating an evaluator: quantum number sum . . . . .	788
7.7.11	Evaluation . . . . .	791
7.7.12	Test . . . . .	792
<b>8</b>	<b>Particles</b>	<b>798</b>
8.1	Polarization . . . . .	798
8.1.1	The polarization type . . . . .	799
8.1.2	Basic initializer and finalizer . . . . .	799
8.1.3	I/O . . . . .	800
8.1.4	Accessing contents . . . . .	801
8.1.5	Initialization from state matrix . . . . .	802
8.1.6	Specific initializers . . . . .	803
8.1.7	Operations . . . . .	810
8.1.8	Test . . . . .	812
8.2	Particles . . . . .	813
8.2.1	The particle type . . . . .	814
8.2.2	Particle sets . . . . .	824
8.2.3	I/O formats . . . . .	829
8.2.4	Expression interface . . . . .	843
8.2.5	Auxiliary stuff . . . . .	844
8.2.6	Test . . . . .	845

<b>9</b>	<b>Initial State</b>	<b>849</b>
9.1	Beams for collisions and decays . . . . .	849
9.1.1	Beam data . . . . .	850
9.1.2	Initializers: collisions . . . . .	854
9.1.3	Initializers: decays . . . . .	856
9.1.4	Data access . . . . .	857
9.1.5	Sanity check . . . . .	857
9.1.6	The beams type . . . . .	858
9.1.7	Inherited procedures . . . . .	860
9.1.8	Accessing contents . . . . .	860
9.1.9	Test . . . . .	860
<b>10</b>	<b>Spectra and structure functions</b>	<b>862</b>
10.1	Tools . . . . .	862
10.1.1	Momentum splitting . . . . .	863
10.1.2	Constant data . . . . .	865
10.1.3	Sampling recoil . . . . .	866
10.1.4	Splitting . . . . .	868
10.1.5	Recovering the splitting . . . . .	870
10.1.6	Extract data . . . . .	871
10.1.7	Unit tests . . . . .	871
10.2	Mappings for structure functions . . . . .	889
10.2.1	Base type . . . . .	889
10.2.2	Integral . . . . .	891
10.2.3	Implementation: standard mapping . . . . .	891
10.2.4	Basic formulas . . . . .	893
10.2.5	Unit tests . . . . .	899
10.3	Structure function base . . . . .	903
10.3.1	Abstract structure-function data type . . . . .	904
10.3.2	Structure-function chain configuration . . . . .	905
10.3.3	Structure-function instance . . . . .	906
10.3.4	Accessing the structure function . . . . .	917
10.3.5	Direct calculations . . . . .	919
10.3.6	Structure-function instance . . . . .	920
10.3.7	Structure-function chain . . . . .	920
10.3.8	Auxiliary stuff . . . . .	923
10.3.9	Chain instances . . . . .	923
10.3.10	Access to the chain instance . . . . .	935
10.3.11	Unit tests . . . . .	937
10.3.12	Test implementation: structure function . . . . .	937
10.3.13	Test implementation: pair spectrum . . . . .	943
10.4	Photon radiation: ISR . . . . .	984
10.4.1	Physics . . . . .	985
10.4.2	Implementation . . . . .	987
10.4.3	The ISR data block . . . . .	987
10.4.4	The ISR object . . . . .	991
10.4.5	Kinematics . . . . .	992
10.4.6	ISR application . . . . .	995
10.4.7	Unit tests . . . . .	997
10.5	EPA . . . . .	1003

10.5.1	Physics	1003
10.5.2	The EPA data block	1004
10.5.3	The EPA object	1008
10.5.4	Kinematics	1009
10.5.5	EPA structure function	1012
10.5.6	EPA application	1013
10.5.7	Unit tests	1014
10.6	EWA	1020
10.6.1	Physics	1020
10.6.2	The EWA data block	1022
10.6.3	The EWA object	1026
10.6.4	Kinematics	1029
10.6.5	EWA structure function	1032
10.6.6	EWA application	1033
10.6.7	Unit tests	1035
10.7	Lepton collider beamstrahlung: CIRCE1	1042
10.7.1	Physics	1042
10.7.2	The CIRCE1 data block	1043
10.7.3	The CIRCE1 object	1046
10.7.4	Kinematics	1048
10.7.5	CIRCE1 structure function	1051
10.7.6	CIRCE1 application	1055
10.8	Lepton collider beamstrahlung: Lumi Linker	1057
10.8.1	Physics	1057
10.8.2	The Lumi Linker data block	1057
10.8.3	The Lumi Linker object	1059
10.8.4	Kinematics	1060
10.8.5	Lumi linker application	1063
10.9	Photon collider: CIRCE2	1065
10.9.1	Physics	1065
10.9.2	The CIRCE2 data block	1065
10.9.3	The CIRCE2 object	1069
10.9.4	Kinematics	1070
10.9.5	CIRCE2 structure function	1073
10.9.6	CIRCE2 application	1075
10.10	Energy-scan spectrum	1077
10.10.1	Data type	1078
10.10.2	The Energy-scan object	1080
10.10.3	Kinematics	1081
10.10.4	Energy-scan structure function	1083
10.10.5	Energy scan application	1084
10.11	Using beam event data	1085
10.11.1	Data type	1085
10.11.2	The beam events object	1088
10.11.3	Kinematics	1089
10.11.4	Beam-event file structure function	1092
10.11.5	Beam events application	1092
10.12	LHAPDF	1094
10.12.1	The module	1094
10.12.2	LHAPDF library interface	1095

10.12.3	The LHAPDF status . . . . .	1096
10.12.4	LHAPDF initialization . . . . .	1097
10.12.5	Kinematics . . . . .	1098
10.12.6	The LHAPDF data block . . . . .	1100
10.12.7	The LHAPDF object . . . . .	1104
10.12.8	Structure function . . . . .	1107
10.12.9	Unit tests . . . . .	1109
10.13	Builtin PDF sets . . . . .	1113
10.13.1	The module . . . . .	1113
10.13.2	The PDF builtin data block . . . . .	1114
10.13.3	The PDF object . . . . .	1117
10.13.4	Kinematics . . . . .	1119
10.13.5	Structure function . . . . .	1120
10.13.6	Strong Coupling . . . . .	1121
10.13.7	Unit tests . . . . .	1122
10.14	User Plugin for Structure Functions . . . . .	1127
10.14.1	The module . . . . .	1127
10.14.2	The user structure function data block . . . . .	1128
10.14.3	The interaction . . . . .	1131
10.14.4	The user structure function . . . . .	1133
10.14.5	Kinematics . . . . .	1134
10.15	Spectra and structure functions: wrapper . . . . .	1136
10.15.1	The structure functions type . . . . .	1137
10.15.2	Mappings . . . . .	1144
10.15.3	Structure function chains . . . . .	1148
10.15.4	Test . . . . .	1162
<b>11</b>	<b>Phase space and hard matrix elements</b>	<b>1166</b>
11.1	Process data block . . . . .	1167
11.2	Abstract phase-space module . . . . .	1168
11.2.1	Phase-space channels . . . . .	1168
11.2.2	Kinematics configuration . . . . .	1171
11.2.3	Extract data . . . . .	1175
11.2.4	Phase-space point instance . . . . .	1177
11.2.5	Auxiliary stuff . . . . .	1184
11.2.6	Unit tests . . . . .	1184
11.3	Single-particle phase space . . . . .	1197
11.3.1	Configuration . . . . .	1197
11.3.2	Kinematics implementation . . . . .	1200
11.3.3	Unit tests . . . . .	1203
11.4	Mappings . . . . .	1209
11.4.1	Default parameters . . . . .	1209
11.4.2	The Mapping type . . . . .	1210
11.4.3	Screen output . . . . .	1211
11.4.4	Define a mapping . . . . .	1212
11.4.5	Retrieve contents . . . . .	1214
11.4.6	Compare mappings . . . . .	1215
11.4.7	Mappings of the invariant mass . . . . .	1215
11.4.8	Step mapping . . . . .	1221
11.4.9	Mappings of the polar angle . . . . .	1224

11.5	Phase-space trees . . . . .	1227
11.5.1	Particles . . . . .	1228
11.5.2	The phase-space tree type . . . . .	1230
11.5.3	PHS tree setup . . . . .	1234
11.5.4	Phase-space evaluation . . . . .	1245
11.6	The phase-space forest . . . . .	1255
11.6.1	Phase-space setup parameters . . . . .	1256
11.6.2	Equivalences . . . . .	1257
11.6.3	Groves . . . . .	1260
11.6.4	The forest type . . . . .	1262
11.6.5	Screen output . . . . .	1264
11.6.6	Accessing contents . . . . .	1267
11.6.7	Read the phase space setup from file . . . . .	1269
11.6.8	Preparation . . . . .	1275
11.6.9	Accessing the particle arrays . . . . .	1276
11.6.10	Find equivalences among phase-space trees . . . . .	1279
11.6.11	Interface for channel equivalences . . . . .	1280
11.6.12	Phase-space evaluation . . . . .	1283
11.6.13	Test of forest setup . . . . .	1287
11.7	Finding phase space parameterizations . . . . .	1289
11.7.1	The mapping modes . . . . .	1290
11.7.2	The cascade type . . . . .	1290
11.7.3	Creating new cascades . . . . .	1297
11.7.4	Tools . . . . .	1298
11.7.5	Hash entries for cascades . . . . .	1299
11.7.6	The cascade set . . . . .	1302
11.7.7	Adding cascades . . . . .	1310
11.7.8	External particles . . . . .	1313
11.7.9	Cascade combination I: flavor assignment . . . . .	1315
11.7.10	Cascade combination II: kinematics setup and check . . .	1316
11.7.11	Cascade combination III: node connections and tree fusion	1322
11.7.12	Cascade set generation . . . . .	1325
11.7.13	Groves . . . . .	1328
11.7.14	Generate the phase space file . . . . .	1329
11.7.15	Test . . . . .	1331
11.8	WOOD phase space . . . . .	1332
11.8.1	Configuration . . . . .	1333
11.8.2	Phase-space generation . . . . .	1335
11.8.3	Phase-space configuration . . . . .	1337
11.8.4	Kinematics implementation . . . . .	1341
11.8.5	Evaluation . . . . .	1342
11.8.6	Unit tests . . . . .	1343
<b>12</b>	<b>Random Number Generation</b>	<b>1359</b>
12.1	Generic Random-Number Generator . . . . .	1359
12.1.1	Generator type . . . . .	1360
12.1.2	Unit tests . . . . .	1361
12.2	TAO Random-Number Generator . . . . .	1364
12.2.1	Generator type . . . . .	1364
12.2.2	Unit tests . . . . .	1366

<b>13 Multi-Channel Integration</b>	<b>1368</b>
13.1 Generic Integrator . . . . .	1368
13.1.1 MCI: integrator . . . . .	1369
13.1.2 MCI instance . . . . .	1376
13.1.3 MCI state . . . . .	1378
13.1.4 MCI sampler . . . . .	1380
13.1.5 Results record . . . . .	1382
13.1.6 Unit tests . . . . .	1383
13.2 Simple midpoint integration . . . . .	1403
13.2.1 Integrator . . . . .	1404
13.2.2 Integrator instance . . . . .	1410
13.2.3 Unit tests . . . . .	1413
13.3 VAMP interface . . . . .	1426
13.3.1 Grid parameters . . . . .	1428
13.3.2 Integration pass . . . . .	1429
13.3.3 Integrator . . . . .	1434
13.3.4 Sampler as Workspace . . . . .	1447
13.3.5 Integrator instance . . . . .	1448
13.3.6 Sampling function . . . . .	1454
13.3.7 Integrator instance: evaluation . . . . .	1455
13.3.8 Unit tests . . . . .	1456
<b>14 Process Libraries</b>	<b>1491</b>
14.1 Process library interface . . . . .	1492
14.1.1 Overview . . . . .	1492
14.1.2 Workflow . . . . .	1492
14.1.3 The module . . . . .	1493
14.1.4 Writers . . . . .	1494
14.1.5 Process records in the library driver . . . . .	1498
14.1.6 The process library driver object . . . . .	1500
14.1.7 Write makefile . . . . .	1504
14.1.8 Write driver file . . . . .	1505
14.1.9 Interface bodies for informational functions . . . . .	1507
14.1.10 Interfaces for C-library matrix element . . . . .	1522
14.1.11 Retrieving the tables . . . . .	1523
14.1.12 Returning a procedure pointer . . . . .	1525
14.1.13 Hooks . . . . .	1527
14.1.14 Make source, compile, link . . . . .	1527
14.1.15 Clean up generated files . . . . .	1528
14.1.16 Further Tools . . . . .	1530
14.1.17 Load the library . . . . .	1531
14.1.18 MD5 sums . . . . .	1534
14.1.19 Test . . . . .	1535
14.2 Abstract process core configuration . . . . .	1562
14.2.1 Process core definition type . . . . .	1563
14.2.2 Process core template . . . . .	1565
14.2.3 Process driver . . . . .	1566
14.2.4 Process driver for intrinsic process . . . . .	1567
14.3 Particle Specifiers . . . . .	1567
14.3.1 The type . . . . .	1568

14.3.2	Constructor . . . . .	1570
14.3.3	Access Methods . . . . .	1571
14.3.4	Test . . . . .	1573
14.4	Process library access . . . . .	1574
14.4.1	Auxiliary stuff . . . . .	1575
14.4.2	Process definition objects . . . . .	1575
14.4.3	Process library . . . . .	1594
14.4.4	Use the library . . . . .	1606
14.4.5	Test . . . . .	1608
14.5	Process Library Stacks . . . . .	1630
14.5.1	The stack entry type . . . . .	1630
14.5.2	The prclib stack type . . . . .	1631
14.5.3	Operating on Stacks . . . . .	1632
14.5.4	Accessing Contents . . . . .	1632
14.5.5	Auxiliary stuff . . . . .	1633
14.5.6	Unit tests . . . . .	1634
14.6	Trivial matrix element for tests . . . . .	1635
14.6.1	Process definition . . . . .	1636
14.6.2	Driver . . . . .	1638
14.6.3	Shortcut . . . . .	1639
14.6.4	Test . . . . .	1640
<b>15</b>	<b>Integration and Event Generation</b>	<b>1644</b>
15.1	Integration results . . . . .	1645
15.1.1	Integration results entry . . . . .	1645
15.1.2	Combined integration results . . . . .	1653
15.1.3	Access results . . . . .	1662
15.1.4	Results display . . . . .	1666
15.2	Abstract process core . . . . .	1670
15.2.1	The process core . . . . .	1671
15.2.2	Storage for intermediate results . . . . .	1676
15.2.3	Helicity selection data . . . . .	1677
15.3	Process observables . . . . .	1677
15.3.1	Abstract base type . . . . .	1679
15.3.2	Initialization . . . . .	1680
15.3.3	Evaluation . . . . .	1681
15.3.4	Implementation for partonic events . . . . .	1682
15.3.5	Implementation for full events . . . . .	1687
15.3.6	Auxiliary stuff . . . . .	1690
15.3.7	Test . . . . .	1690
15.4	Parton states . . . . .	1696
15.4.1	Abstract base type . . . . .	1697
15.4.2	Common Initialization . . . . .	1700
15.4.3	Evaluator initialization: isolated state . . . . .	1701
15.4.4	Evaluator initialization: connected state . . . . .	1703
15.4.5	Cuts and expressions . . . . .	1704
15.4.6	Evaluation . . . . .	1706
15.4.7	Accessing the state . . . . .	1708
15.4.8	Auxiliary stuff . . . . .	1709
15.5	Complete Elementary Processes . . . . .	1710

15.5.1	The Process Object . . . . .	1712
15.5.2	Metadata . . . . .	1726
15.5.3	Generic Configuration Data . . . . .	1728
15.5.4	Beam configuration . . . . .	1732
15.5.5	Multi-channel integration . . . . .	1735
15.5.6	Process Components . . . . .	1741
15.5.7	Process terms . . . . .	1744
15.5.8	Default Iterations . . . . .	1748
15.5.9	Constant process data . . . . .	1750
15.5.10	Compute an amplitude . . . . .	1754
15.5.11	Process instances . . . . .	1755
15.5.12	Accessing the process instance . . . . .	1790
15.5.13	Particle sets . . . . .	1793
15.5.14	Auxiliary stuff . . . . .	1794
15.5.15	Unit tests . . . . .	1794
15.6	Process Stacks . . . . .	1827
15.6.1	The process entry type . . . . .	1828
15.6.2	The process stack type . . . . .	1828
15.6.3	Push . . . . .	1829
15.6.4	Data Access . . . . .	1830
15.6.5	Auxiliary stuff . . . . .	1830
15.6.6	Unit tests . . . . .	1830
15.7	Complete Events . . . . .	1832
15.7.1	Event configuration . . . . .	1833
15.7.2	The event type . . . . .	1834
15.7.3	Initialization . . . . .	1836
15.7.4	Evaluation . . . . .	1838
15.7.5	Reset to empty state . . . . .	1839
15.7.6	Generation . . . . .	1839
15.7.7	Recovering an event . . . . .	1840
15.7.8	Access content . . . . .	1842
15.7.9	Auxiliary stuff . . . . .	1845
15.7.10	Unit tests . . . . .	1845
<b>16</b>	<b>Matrix Element Implementations</b>	<b>1853</b>
16.1	O'MEGA Interface . . . . .	1853
16.1.1	Process definition . . . . .	1854
16.1.2	The O'MEGA writer . . . . .	1858
16.1.3	Driver . . . . .	1864
16.1.4	High-level process definition . . . . .	1864
16.1.5	The <code>prc_omega_t</code> wrapper . . . . .	1865
16.1.6	Test . . . . .	1872
<b>17</b>	<b>Event I/O</b>	<b>1890</b>
17.1	Event Sample Data . . . . .	1890
17.1.1	Event Sample Data . . . . .	1890
17.1.2	Unit tests . . . . .	1892
17.2	Abstract I/O Handler . . . . .	1893
17.2.1	Process pointer . . . . .	1894
17.2.2	Type . . . . .	1894



17.2.3	Unit tests . . . . .	1897
17.3	Raw Event I/O . . . . .	1902
17.3.1	Type . . . . .	1903
17.3.2	Unit tests . . . . .	1907
17.4	HEP Common Blocks . . . . .	1910
17.4.1	Event characteristics . . . . .	1911
17.4.2	Particle characteristics . . . . .	1911
17.4.3	The HEPUP common block . . . . .	1912
17.4.4	Run parameter output (verbose) . . . . .	1914
17.4.5	Run parameter output (other formats) . . . . .	1915
17.4.6	The HEPEUP common block . . . . .	1915
17.4.7	The HEPEVT common block . . . . .	1918
17.4.8	Event output . . . . .	1921
17.4.9	Event output in various formats . . . . .	1923
17.4.10	Data Transfer: particle sets . . . . .	1925
17.4.11	Data Transfer: events . . . . .	1927
17.5	LHEF Input/Output . . . . .	1928
17.5.1	Type . . . . .	1929
17.5.2	Specific Methods . . . . .	1929
17.5.3	Common Methods . . . . .	1929
17.5.4	Les Houches Event File: header/footer . . . . .	1932
17.5.5	Unit tests . . . . .	1932
17.6	HepMC events . . . . .	1935
17.6.1	Interface check . . . . .	1935
17.6.2	FourVector . . . . .	1936
17.6.3	Polarization . . . . .	1938
17.6.4	GenParticle . . . . .	1941
17.6.5	GenVertex . . . . .	1948
17.6.6	Vertex-particle-in iterator . . . . .	1951
17.6.7	Vertex-particle-out iterator . . . . .	1953
17.6.8	GenEvent . . . . .	1956
17.6.9	Event-particle iterator . . . . .	1964
17.6.10	I/O streams . . . . .	1966
17.6.11	Test . . . . .	1968
17.7	HepMC Output . . . . .	1970
17.7.1	Type . . . . .	1971
17.7.2	Specific Methods . . . . .	1971
17.7.3	Common Methods . . . . .	1972
17.7.4	Particle Set Transfer . . . . .	1974
17.7.5	Unit tests . . . . .	1976
17.8	Event Weight Output . . . . .	1978
17.8.1	Type . . . . .	1979
17.8.2	Unit tests . . . . .	1982
<b>18</b>	<b>The SUSY Les Houches Accord</b>	<b>1984</b>
18.0.3	Preprocessor . . . . .	1985
18.0.4	Lexer and syntax . . . . .	1986
18.0.5	Interpreter . . . . .	1988
18.0.6	Auxiliary function . . . . .	1994
18.0.7	Parser . . . . .	2006

18.0.8	API	2007
18.0.9	Test	2008
<b>19</b>	<b>Integration and simulation</b>	<b>2010</b>
19.1	Iterations	2010
19.1.1	The iterations list	2011
19.1.2	Tools	2013
19.1.3	Test	2016
19.2	Beam polarization	2018
19.2.1	Parameters and type definition	2019
19.2.2	Constructors	2019
19.2.3	Tools	2021
19.3	Beam structure	2025
19.3.1	Beam structure elements	2026
19.3.2	Beam structure type	2026
19.3.3	Get contents	2030
19.3.4	Unit Tests	2031
19.4	User-controlled File I/O	2036
19.4.1	The file type	2036
19.4.2	The file list	2039
19.5	Runtime data	2042
19.5.1	Container for parse nodes	2043
19.5.2	The data type	2044
19.5.3	Output	2045
19.5.4	Initialization	2047
19.5.5	Local copies	2060
19.5.6	Finalization	2063
19.5.7	Filling Contents	2063
19.5.8	Get contents	2064
19.5.9	Auxiliary stuff	2065
19.5.10	Test	2065
19.6	Select implementations	2071
19.6.1	Process Core Definition	2073
19.6.2	Process core allocation	2074
19.6.3	Process core update and restoration	2074
19.6.4	Integrator allocation	2075
19.6.5	Phase-space allocation	2076
19.6.6	Random-number generator	2078
19.6.7	Structure function configuration data	2078
19.6.8	Event I/O stream	2082
19.6.9	QCD coupling	2083
19.6.10	Test	2084
19.7	Process Configuration	2099
19.7.1	Data Type	2100
19.7.2	Test	2102
19.8	Compilation	2107
19.8.1	The data type	2108
19.8.2	API for compilation and loading	2110
19.8.3	Test	2112
19.9	Integration	2113

19.9.1	The integration type . . . . .	2115
19.9.2	Initialization . . . . .	2116
19.9.3	Integration . . . . .	2126
19.9.4	API for integration objects . . . . .	2131
19.9.5	Test . . . . .	2139
19.10	Event Streams . . . . .	2151
19.10.1	Event Stream Array . . . . .	2152
19.10.2	Unit Tests . . . . .	2156
19.11	Simulation . . . . .	2162
19.11.1	Event counting . . . . .	2163
19.11.2	Simulation: component sets . . . . .	2164
19.11.3	Process-core Safe . . . . .	2166
19.11.4	Simulation entry . . . . .	2166
19.11.5	The simulation type . . . . .	2171
19.11.6	Event Stream I/O . . . . .	2179
19.11.7	Event Stream Array . . . . .	2181
19.11.8	Recover event . . . . .	2182
19.11.9	Auxiliary stuff . . . . .	2182
19.11.10	Extract contents . . . . .	2183
19.11.11	Test . . . . .	2184
<b>20</b>	<b>Top level API</b>	<b>2206</b>
20.1	Commands . . . . .	2206
20.1.1	The command type . . . . .	2207
20.1.2	Specific command types . . . . .	2212
20.2	User-controlled output to data files . . . . .	2300
20.2.1	Print default-formatted values . . . . .	2302
20.2.2	Parameters: random-number generator seed . . . . .	2322
20.2.3	The command list . . . . .	2340
20.2.4	Compiling the parse tree . . . . .	2341
20.2.5	Executing the command list . . . . .	2342
20.2.6	Command list syntax . . . . .	2342
20.2.7	Unit Tests . . . . .	2350
20.3	Toplevel module WHIZARD . . . . .	2374
20.3.1	Options . . . . .	2376
20.3.2	The <code>whizard</code> object . . . . .	2376
20.3.3	Initialization and finalization . . . . .	2377
20.3.4	Initialization and finalization (old version) . . . . .	2379
20.3.5	Execute command lists . . . . .	2383
20.3.6	The WHIZARD shell . . . . .	2385
20.3.7	Self-tests . . . . .	2386
20.4	Driver program . . . . .	2389
20.5	Shower . . . . .	2399
20.6	Whizard-C-Interface . . . . .	2425
<b>21</b>	<b>Cross References</b>	<b>2435</b>
21.1	Identifiers . . . . .	2435
21.2	Chunks . . . . .	2435

# Chapter 1

# Changes

For a comprehensive list of changes confer the ChangeLog file or the subversion log.

# Chapter 2

## Tools

This part contains modules needed for auxiliary purposes:

**limits** Compile-time (integer) constants: fixed array sizes, input field lengths, and such. Any module that uses such constants has to access them via **limits**.

**mpi90** Parallel execution (currently disabled!). This module contains dummy replacements for the message passing interface subroutines.

**clock** Store and handle dates and times

**diagnostics** Error and diagnostic message handling. Any messages and errors issued by WHIZARD functions are handled by the subroutines in this module, if possible.

**files** Files and auxiliary routines that do not belong anywhere else.

**permutations** Handle permutations of integers.

**unix\_args** Parse the command-line arguments and options. This part is system-dependent, and we provide a replacement if the functions are not available.

### 2.1 Preliminaries

The WHIZARD file header:

```
<File header>≡
! WHIZARD <Version> <Date>
!
! Copyright (C) 1999-2013 by
!   Wolfgang Kilian <kilian@physik.uni-siegen.de>
!   Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
!   Juergen Reuter <juergen.reuter@desy.de>
!
!   with contributions by
!   Christian Speckner <cnspeckn@googlemail.com>
!   and Felix Braam, Sebastian Schmidt, Daniel Wiesler
!
! WHIZARD is free software; you can redistribute it and/or modify it
```

```

! under the terms of the GNU General Public License as published by
! the Free Software Foundation; either version 2, or (at your option)
! any later version.
!
! WHIZARD is distributed in the hope that it will be useful, but
! WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This file has been stripped of most comments. For documentation, refer
! to the source 'whizard.nw'

```

We are strict with our names:

```

<Standard module head>≡
    implicit none
    private

```

This is the way to invoke the kinds module (not contained in this source)

```

<Use kinds>≡
    use kinds, only: default !NODEP!

```

And we make heavy use of variable-length strings

```

<Use strings>≡
    use iso_varying_string, string_t => varying_string !NODEP!

```

Some parameters (buffer sizes etc.) are hardcoded. They are collected in this module:

```

<limits.f90>≡
    <File header>

    module limits

    use iso_fortran_env, only: iostat_end, iostat_eor !NODEP!
    <Use kinds>

    <Standard module head>

    <Limits: public parameters>

    end module limits

```

The version string is used for checking files. Note that the string length MUST NOT be changed, because reading binary files relies on it.

```

<Limits: public parameters>≡
    integer, parameter, public :: VERSION_STRLEN = 255
    character(len=VERSION_STRLEN), parameter, public :: &
        & VERSION_STRING = "WHIZARD version <Version> (<Date>)"

```

## 2.2 Parallelization

*This section has been removed, it was never used even in WHIZARD1. Parallelization should be reimplemented from scratch.*

## 2.3 File utilities

This module provides miscellaneous tools associated with strings and files:

- Finding a free unit
- Selecting an output unit (stdout if undefined)
- Upper and lower case for strings
- Formatting a number for T<sub>E</sub>X output.

```
<file_utils.f90>≡  
<File header>  
  
module file_utils  
  
    use iso_fortran_env, only: stdout => output_unit !NODEP!  
    <Use kinds>  
    <Use strings>  
    use limits, only: MIN_UNIT, MAX_UNIT !NODEP!  
    use, intrinsic :: iso_c_binding !NODEP!  
  
    <Standard module head>  
  
    <File utils: public>  
  
    <File utils: interfaces>  
  
contains  
  
    <File utils: procedures>  
  
end module file_utils
```

### 2.3.1 Finding an I/O unit

Fortran 95 (even Fortran 2003) has no notion of implicit I/O units. Therefore, we have to find a free unit by trial and error.

```
<Limits: public parameters>+≡  
    integer, parameter, public :: MIN_UNIT = 11, MAX_UNIT = 99  
  
<File utils: public>≡  
    public :: free_unit
```

```

<File utils: procedures>≡
function free_unit () result (unit)
  integer :: unit
  logical :: exists, is_open
  integer :: i, status
  do i = MIN_UNIT, MAX_UNIT
    inquire (unit=i, exist=exists, opened=is_open, iostat=status)
    if (status == 0) then
      if (exists .and. .not. is_open) then
        unit = i; return
      end if
    end if
  end do
  unit = -1
end function free_unit

```

```

<Use file utils>≡
use file_utils !NODEP!

```

Return the given unit, if present, otherwise the default STDOUT unit.

```

<File utils: public>+≡
public :: output_unit

<File utils: procedures>+≡
function output_unit (unit) result (u)
  integer, intent(in), optional :: unit
  integer :: u
  if (present (unit)) then
    u = unit
  else
    u = stdout
  end if
end function output_unit

```

### 2.3.2 Deleting a file

Fortran does not contain a command for deleting a file. Here, we provide a subroutine that deletes a file if it exists. We do not handle the subtleties, so we assume that it is writable if it exists.

```

<File utils: public>+≡
public :: delete_file

<File utils: procedures>+≡
subroutine delete_file (name)
  character(*), intent(in) :: name
  logical :: exist
  integer :: u
  inquire (file = name, exist = exist)
  if (exist) then
    u = free_unit ()
    open (unit = u, file = name)
    close (u, status = "delete")
  end if
end subroutine delete_file

```



```
end subroutine delete_file
```

### 2.3.3 String auxiliary functions

These are, unfortunately, not part of Fortran.

*<File utils: public>+≡*

```
public :: upper_case
public :: lower_case
```

*<File utils: interfaces>≡*

```
interface upper_case
  module procedure upper_case_char, upper_case_string
end interface
interface lower_case
  module procedure lower_case_char, lower_case_string
end interface
```

*<File utils: procedures>+≡*

```
function upper_case_char (string) result (new_string)
  character(*), intent(in) :: string
  character(len(string)) :: new_string
  integer :: pos, code
  integer, parameter :: offset = ichar('A')-ichar('a')
  do pos = 1, len (string)
    code = ichar (string(pos:pos))
    select case (code)
      case (ichar('a'):ichar('z'))
        new_string(pos:pos) = char (code + offset)
      case default
        new_string(pos:pos) = string(pos:pos)
    end select
  end do
end function upper_case_char

function lower_case_char (string) result (new_string)
  character(*), intent(in) :: string
  character(len(string)) :: new_string
  integer :: pos, code
  integer, parameter :: offset = ichar('a')-ichar('A')
  do pos = 1, len (string)
    code = ichar (string(pos:pos))
    select case (code)
      case (ichar('A'):ichar('Z'))
        new_string(pos:pos) = char (code + offset)
      case default
        new_string(pos:pos) = string(pos:pos)
    end select
  end do
end function lower_case_char

function upper_case_string (string) result (new_string)
  type(string_t), intent(in) :: string
  type(string_t) :: new_string
```

```

    new_string = upper_case_char (char (string))
end function upper_case_string

function lower_case_string (string) result (new_string)
    type(string_t), intent(in) :: string
    type(string_t) :: new_string
    new_string = lower_case_char (char (string))
end function lower_case_string

```

Quote underscore characters for use in T<sub>E</sub>X output.

```

<File utils: public>+≡
    public :: quote_underscore

<File utils: procedures>+≡
    function quote_underscore (string) result (quoted)
        type(string_t) :: quoted
        type(string_t), intent(in) :: string
        type(string_t) :: part
        type(string_t) :: buffer
        buffer = string
        quoted = ""
        do
            call split (part, buffer, "_")
            quoted = quoted // part
            if (buffer == "") exit
            quoted = quoted // "\"
        end do
    end function quote_underscore

```

Convert a FORTRAN string into a zero terminated C string.

```

<File utils: public>+≡
    public :: string_f2c

<File utils: interfaces>+≡
    interface string_f2c
        module procedure string_f2c_char, string_f2c_var_str
    end interface string_f2c

<File utils: procedures>+≡
    pure function string_f2c_char (i) result (o)
        character(*), intent(in) :: i
        character(kind=c_char, len=len (i) + 1) :: o
        o = i // c_null_char
    end function string_f2c_char

    pure function string_f2c_var_str (i) result (o)
        type(string_t), intent(in) :: i
        character(kind=c_char, len=len (i) + 1) :: o
        o = char (i) // c_null_char
    end function string_f2c_var_str

```

### 2.3.4 Formatting numbers

Format a number with  $n$  significant digits for use in T<sub>E</sub>X documents.

*(File utils: public)*+≡

public :: tex\_format

*(File utils: procedures)*+≡

```
function tex_format (rval, n_digits) result (string)
  type(string_t) :: string
  real(default), intent(in) :: rval
  integer, intent(in) :: n_digits
  integer :: e, n, w, d
  real(default) :: absval
  real(default) :: mantissa
  character :: sign
  character(20) :: format
  character(80) :: cstr
  n = min (abs (n_digits), 16)
  if (rval == 0) then
    string = "0"
  else
    absval = abs (rval)
    e = log10 (absval)
    if (rval < 0) then
      sign = "-"
    else
      sign = ""
    end if
    select case (e)
    case (:-3)
      d = max (n - 1, 0)
      w = max (d + 2, 2)
      write (format, "('(F',IO,'.',IO,'A,IO,A)')") w, d
      mantissa = absval * 10._default ** (1 - e)
      write (cstr, fmt=format) mantissa, "\times 10^{", e - 1, "}"
    case (-2:0)
      d = max (n - e, 1 - e)
      w = max (d + e + 2, d + 2)
      write (format, "('(F',IO,'.',IO,')')") w, d
      write (cstr, fmt=format) absval
    case (1:2)
      d = max (n - e - 1, -e, 0)
      w = max (d + e + 2, d + 2, e + 2)
      write (format, "('(F',IO,'.',IO,')')") w, d
      write (cstr, fmt=format) absval
    case default
      d = max (n - 1, 0)
      w = max (d + 2, 2)
      write (format, "('(F',IO,'.',IO,'A,IO,A)')") w, d
      mantissa = absval * 10._default ** (- e)
      write (cstr, fmt=format) mantissa, "\times 10^{", e, "}"
    end select
    string = sign // trim (cstr)
  end if
end function tex_format
```

Write a number for use in Metapost code:

```

<File utils: public>+=
  public :: mp_format
<File utils: procedures>+=
  function mp_format (rval) result (string)
    type(string_t) :: string
    real(default), intent(in) :: rval
    character(16) :: tmp
    write (tmp, "(G16.8)")  rval
    string = lower_case (trim (adjustl (trim (tmp))))
  end function mp_format

```

## 2.4 Unit tests

Here, we provide functionality for automated unit tests. Each test is required to produce output which is compared against a reference file. If the two are identical, we signal success. Otherwise, we signal failure and write the output to a file.

```

<unit_tests.f90>=
  module unit_tests

    <Use strings>
    <Use file utils>

    <Standard module head>

    <Tests: public>

    <Tests: parameters>

    <Tests: types>

    <Tests: interfaces>

    contains

    <Tests: procedures>

  end module unit_tests

```

### 2.4.1 Parameters

Building blocks of file names:

```

<Tests: parameters>=
  character(*), parameter :: ref_prefix = "ref-output/"
  character(*), parameter :: ref = ".ref"

  character(*), parameter :: err_prefix = "err-output/"

```

```
character(*), parameter :: err = ".out"
```

## 2.4.2 Type for storing test results

We store the results of the individual unit tests in a linked list. Here is the entry:

```
<Tests: public>≡
    public :: test_results_t

<Tests: types>≡
    type :: test_result_t
        logical :: success = .false.
        type(string_t) :: name
        type(string_t) :: description
        type(test_result_t), pointer :: next => null ()
    end type test_result_t

    type :: test_results_t
        private
        type(test_result_t), pointer :: first => null ()
        type(test_result_t), pointer :: last => null ()
        integer :: n_success = 0
        integer :: n_failure = 0
        contains
        <Tests: test results: TBP>
    end type test_results_t
```

Add a test result.

```
<Tests: test results: TBP>≡
    procedure, private :: add => test_results_add

<Tests: procedures>≡
    subroutine test_results_add (list, name, description, success)
        class(test_results_t), intent(inout) :: list
        character(len=*), intent(in) :: name
        character(len=*), intent(in) :: description
        logical, intent(in) :: success
        type(test_result_t), pointer :: result
        allocate (result)
        result%success = success
        result%name = name
        result%description = description
        if (associated (list%first)) then
            list%last%next => result
        else
            list%first => result
        end if
        list%last => result
        if (success) then
            list%n_success = list%n_success + 1
        else
            list%n_failure = list%n_failure + 1
        end if
    end subroutine
```

```
end subroutine test_results_add
```

Display the current state.

*<Tests: test results: TBP>+≡*

```
procedure, private :: write => test_results_write
```

*<Tests: procedures>+≡*

```
subroutine test_results_write (list, u)
  class(test_results_t), intent(in) :: list
  integer, intent(in) :: u
  type(test_result_t), pointer :: result
  write (u, "(A)")  "*** Test Summary ***"
  if (list%n_success > 0) then
    write (u, "(2x,A)") "Success:"
    result => list%first
    do while (associated (result))
      if (result%success) write (u, "(4x,A,': ',A)") &
        char (result%name), char (result%description)
      result => result%next
    end do
  end if
  if (list%n_failure > 0) then
    write (u, "(2x,A)") "Failure:"
    result => list%first
    do while (associated (result))
      if (.not. result%success) write (u, "(4x,A,': ',A)") &
        char (result%name), char (result%description)
      result => result%next
    end do
  end if
  write (u, "(A,I0)") "Total   = ", list%n_success + list%n_failure
  write (u, "(A,I0)") "Success = ", list%n_success
  write (u, "(A,I0)") "Failure = ", list%n_failure
  write (u, "(A)")  "*** End of test Summary ***"
end subroutine test_results_write
```

Return true if all tests were successful (or no test).

*<Tests: test results: TBP>+≡*

```
procedure, private :: report => test_results_report
```

*<Tests: procedures>+≡*

```
subroutine test_results_report (list, success)
  class(test_results_t), intent(in) :: list
  logical, intent(out) :: success
  success = list%n_failure == 0
end subroutine test_results_report
```

Delete the list.

*<Tests: test results: TBP>+≡*

```
procedure, private :: final => test_results_final
```

*<Tests: procedures>+≡*

```
subroutine test_results_final (list)
  class(test_results_t), intent(inout) :: list
```

```

type(test_result_t), pointer :: result
do while (associated (list%first))
    result => list%first
    list%first => result%next
    deallocate (result)
end do
list%last => null ()
list%n_success = 0
list%n_failure = 0
end subroutine test_results_final

```

### 2.4.3 Wrapup

This will write results, report status, and finalize. This is the only method which we need to access from outside.

```

<Tests: test results: TBP>+≡
    procedure :: wrapup => test_results_wrapup

<Tests: procedures>+≡
    subroutine test_results_wrapup (list, u, success)
        class(test_results_t), intent(inout) :: list
        integer, intent(in) :: u
        logical, intent(out), optional :: success
        call list%write (u)
        if (present (success)) call list%report (success)
        call list%final ()
    end subroutine test_results_wrapup

```

### 2.4.4 Tool for Unit Tests

This procedure takes a test routine as an argument. It runs the test, output directed to a temporary file. Then, it compares the file against a reference file.

The test routine must take the output unit as argument.

```

<Tests: interfaces>≡
    abstract interface
        subroutine unit_test (u)
            integer, intent(in) :: u
        end subroutine unit_test
    end interface

```

The test routine can print to screen and, optionally, to a logging unit.

```

<Tests: public>+≡
    public :: test

<Tests: procedures>+≡
    subroutine test (test_proc, name, description, u_log, results)
        procedure(unit_test) :: test_proc
        character(*), intent(in) :: name
        character(*), intent(in) :: description
        integer, intent(in) :: u_log
        type(test_results_t), intent(inout) :: results
    end subroutine test

```

```

integer :: u_test, u_ref, u_err
logical :: exist
character(256) :: buffer1, buffer2
integer :: iostat1, iostat2
logical :: success
write (*, "(A)", advance="no") "Running test: " // name
write (u_log, "(A)") "Test: " // name
u_test = free_unit ()
open (u_test, status="scratch", action="readwrite")
call test_proc (u_test)
rewind (u_test)
inquire (file=ref_prefix//name//ref, exist=exist)
if (exist) then
    u_ref = free_unit ()
    open (u_ref, file=ref_prefix//name//ref, status="old", action="read")
    COMPARE_FILES: do
        read (u_test, "(A)", iostat=iostat1) buffer1
        read (u_ref, "(A)", iostat=iostat2) buffer2
        if (iostat1 /= iostat2) then
            success = .false.
            exit COMPARE_FILES
        else if (iostat1 < 0) then
            success = .true.
            exit COMPARE_FILES
        else if (buffer1 /= buffer2) then
            success = .false.
            exit COMPARE_FILES
        end if
    end do COMPARE_FILES
    close (u_ref)
else
    write (*, "(A)", advance="no") " ... no reference output available"
    write (u_log, "(A)") " No reference output available."
    success = .false.
end if
if (success) then
    write (*, "(A)") " ... success."
    write (u_log, "(A)") " Success."
else
    write (*, "(A)") " ... failure. See: " // err_prefix//name//err
    write (u_log, "(A)") " Failure."
    rewind (u_test)
    u_err = free_unit ()
    open (u_err, file=err_prefix//name//err, &
        action="write", status="replace")
    WRITE_OUTPUT: do
        read (u_test, "(A)", end=1) buffer1
        write (u_err, "(A)") trim (buffer1)
    end do WRITE_OUTPUT
1    close (u_err)
end if
close (u_test)
call results%add (name, description, success)
end subroutine test

```



## 2.5 Operating-system interface

For specific purposes, we need direct access to the OS (system calls). This is, of course, system dependent. The current version is valid for GNU/Linux; we expect to use a preprocessor for this module if different OSs are to be supported.

The current implementation lacks error handling.

```
<os_interface.f90>≡
  <File header>

  module os_interface

    use iso_c_binding !NODEP!
    <Use strings>
    <Use file utils>
    use system_dependencies !NODEP!
    use limits, only: DLERROR_LEN, ENVVAR_LEN !NODEP!
    use diagnostics !NODEP!

    <Standard module head>

    <OS interface: public>

    <OS interface: types>

    <OS interface: interfaces>

    contains

    <OS interface: procedures>

  end module os_interface
```

### 2.5.1 Path variables

This is a transparent container for storing user-defined path variables.

```
<OS interface: public>≡
  public :: paths_t
<OS interface: types>≡
  type :: paths_t
    type(string_t) :: prefix
    type(string_t) :: exec_prefix
    type(string_t) :: bindir
    type(string_t) :: libdir
    type(string_t) :: includedir
    type(string_t) :: datarootdir
    type(string_t) :: localprefix
    type(string_t) :: libtool
    type(string_t) :: lhapdfdir
  end type paths_t
```

```

<OS interface: public>+≡
    public :: paths_init

<OS interface: procedures>≡
    subroutine paths_init (paths)
        type(paths_t), intent(out) :: paths
        paths%prefix = ""
        paths%exec_prefix = ""
        paths%bindir = ""
        paths%libdir = ""
        paths%includedir = ""
        paths%datarootdir = ""
        paths%localprefix = ""
        paths%libtool = ""
        paths%lhapdfdir = ""
    end subroutine paths_init

```

## 2.5.2 System dependencies

We store all potentially system- and user/run-dependent data in a transparent container. This includes compiler/linker names and flags, file extensions, etc.

```

<OS interface: public>+≡
    public :: os_data_t

<OS interface: types>+≡
    type :: os_data_t
        logical :: use_libtool
        logical :: use_testfiles
        type(string_t) :: fc
        type(string_t) :: fcflags
        type(string_t) :: fcflags_pic
        type(string_t) :: fc_src_ext
        type(string_t) :: cc
        type(string_t) :: cflags
        type(string_t) :: cflags_pic
        type(string_t) :: obj_ext
        type(string_t) :: ld
        type(string_t) :: ldflags
        type(string_t) :: ldflags_so
        type(string_t) :: ldflags_static
        type(string_t) :: ldflags_hepmc
        type(string_t) :: shlib_ext
        type(string_t) :: makeflags
        type(string_t) :: prefix
        type(string_t) :: exec_prefix
        type(string_t) :: bindir
        type(string_t) :: libdir
        type(string_t) :: includedir
        type(string_t) :: datarootdir
        type(string_t) :: whizard_omega_binpath
        type(string_t) :: whizard_includes
        type(string_t) :: whizard_ldflags
        type(string_t) :: whizard_libtool

```

```

type(string_t) :: whizard_modelpath
type(string_t) :: whizard_models_libpath
type(string_t) :: whizard_susypath
type(string_t) :: whizard_gmlpath
type(string_t) :: whizard_cutspath
type(string_t) :: whizard_texpath
type(string_t) :: whizard_testdatapath
type(string_t) :: whizard_modelpath_local
type(string_t) :: whizard_models_libpath_local
type(string_t) :: whizard_omega_binpath_local
type(string_t) :: whizard_circe2path
type(string_t) :: whizard_beamsimpath
type(string_t) :: whizard_mulipath
type(string_t) :: pdf_builtin_datapath
logical :: event_analysis = .false.
logical :: event_analysis_ps = .false.
logical :: event_analysis_pdf = .false.
type(string_t) :: latex
type(string_t) :: mpost
type(string_t) :: gml
type(string_t) :: dvips
type(string_t) :: ps2pdf
end type os_data_t

```

Since all are allocatable strings, explicit initialization is necessary.

```

<Limits: public parameters>+≡
integer, parameter, public :: ENVVAR_LEN = 1000

<OS interface: public>+≡
public :: os_data_init

<OS interface: procedures>+≡
subroutine os_data_init (os_data, paths)
type(os_data_t), intent(out) :: os_data
type(paths_t), intent(in), optional :: paths
character(len=ENVVAR_LEN) :: home
type(string_t) :: localprefix, local_includes
os_data%use_libtool = .true.
inquire (file = "TESTFLAG", exist = os_data%use_testfiles)
call get_environment_variable ("HOME", home)
if(present(paths)) then
if (paths%localprefix == "") then
localprefix = trim (home) // "/.whizard"
else
localprefix = paths%localprefix
end if
else
localprefix = trim (home) // "/.whizard"
end if
local_includes = localprefix // "/lib/whizard/mod/models"
os_data%whizard_modelpath_local = localprefix // "/share/whizard/models"
os_data%whizard_models_libpath_local = localprefix // "/lib/whizard/models"
os_data%whizard_omega_binpath_local = localprefix // "/bin"
os_data%fc = DEFAULT_FC
os_data%fcflags = DEFAULT_FCFLAGS

```

```

os_data%fcflags_pic      = DEFAULT_FCFLAGS_PIC
os_data%fc_src_ext       = DEFAULT_FC_SRC_EXT
os_data%cc               = DEFAULT_CC
os_data%cflags           = DEFAULT_CFLAGS
os_data%cflags_pic       = DEFAULT_CFLAGS_PIC
os_data%obj_ext          = DEFAULT_OBJ_EXT
os_data%ld               = DEFAULT_LD
os_data%ldflags          = DEFAULT_LDFLAGS
os_data%ldflags_so       = DEFAULT_LDFLAGS_SO
os_data%ldflags_static   = DEFAULT_LDFLAGS_STATIC
os_data%ldflags_hepmc    = DEFAULT_LDFLAGS_HEPMC
os_data%shlib_ext        = DEFAULT_SHLIB_EXT
os_data%makeflags        = DEFAULT_MAKEFLAGS
os_data%prefix           = PREFIX
os_data%exec_prefix      = EXEC_PREFIX
os_data%bindir           = BINDIR
os_data%libdir           = LIBDIR
os_data%includedir       = INCLUDEDIR
os_data%datarootdir      = DATAROOTDIR
if (present (paths)) then
  if (paths%prefix /= "") os_data%prefix = paths%prefix
  if (paths%exec_prefix /= "") os_data%exec_prefix = paths%exec_prefix
  if (paths%bindir /= "") os_data%bindir = paths%bindir
  if (paths%libdir /= "") os_data%libdir = paths%libdir
  if (paths%includedir /= "") os_data%includedir = paths%includedir
  if (paths%datarootdir /= "") os_data%datarootdir = paths%datarootdir
end if
if (os_data%use_testfiles) then
  os_data%whizard_omega_binpath = WHIZARD_TEST_OMEGA_BINPATH
  os_data%whizard_includes      = WHIZARD_TEST_INCLUDES
  os_data%whizard_ldflags       = WHIZARD_TEST_LDFLAGS
  os_data%whizard_libtool       = WHIZARD_LIBTOOL_TEST
  os_data%whizard_modelpath     = WHIZARD_TEST_MODELPATH
  os_data%whizard_models_libpath = WHIZARD_TEST_MODELS_LIBPATH
  os_data%whizard_susypath      = WHIZARD_TEST_SUSYPATH
  os_data%whizard_gmlpath       = WHIZARD_TEST_GMLPATH
  os_data%whizard_cutspath      = WHIZARD_TEST_CUTSPATH
  os_data%whizard_texpath       = WHIZARD_TEST_TEXPATH
  os_data%whizard_testdatapath  = WHIZARD_TEST_TESTDATAPATH
  os_data%whizard_circe2path    = WHIZARD_TEST_CIRCE2PATH
  os_data%whizard_beamsimpath   = WHIZARD_TEST_BEAMSIMPATH
  os_data%whizard_mulipath      = WHIZARD_TEST_MULIPATH
  os_data%pdf_builtin_datapath  = PDF_BUILTIN_TEST_DATAPATH
else
  if (os_dir_exist (local_includes)) then
    os_data%whizard_includes = "-I" // local_includes // " " // &
    WHIZARD_INCLUDES
  else
    os_data%whizard_includes = WHIZARD_INCLUDES
  end if
  os_data%whizard_omega_binpath = WHIZARD_OMEGA_BINPATH
  os_data%whizard_ldflags       = WHIZARD_LDFLAGS
  os_data%whizard_libtool       = WHIZARD_LIBTOOL
  if (present(paths)) then

```

```

        if (paths%libtool /= "") os_data%whizard_libtool = paths%libtool
    end if
    os_data%whizard_modelpath      = WHIZARD_MODELPATH
    os_data%whizard_models_libpath = WHIZARD_MODELS_LIBPATH
    os_data%whizard_susypath       = WHIZARD_SUSYPATH
    os_data%whizard_gmlpath        = WHIZARD_GMLPATH
    os_data%whizard_cutspath       = WHIZARD_CUTSPATH
    os_data%whizard_texpath        = WHIZARD_TEXPATH
    os_data%whizard_testdatapath   = WHIZARD_TESTDATAPATH
    os_data%whizard_circe2path     = WHIZARD_CIRCE2PATH
    os_data%whizard_beamsimpath    = WHIZARD_BEAMSIMPATH
    os_data%whizard_mulipath       = WHIZARD_MULIPATH
    os_data%pdf_builtin_datapath   = PDF_BUILTIN_DATAPATH
end if
os_data%event_analysis      = EVENT_ANALYSIS      == "yes"
os_data%event_analysis_ps   = EVENT_ANALYSIS_PS   == "yes"
os_data%event_analysis_pdf  = EVENT_ANALYSIS_PDF  == "yes"
os_data%latex               = PRG_LATEX // " " // OPT_LATEX
os_data%mpost               = PRG_MPOST // " " // OPT_MPOST
os_data%gml                 = os_data%whizard_gmlpath // "/gml" // " " // OPT_MPOST &
                             // " " // "--gml_dir " // os_data%whizard_gmlpath
os_data%dvips               = PRG_DVIPS
os_data%ps2pdf               = PRG_PS2PDF
call os_data_expand_paths (os_data)
end subroutine os_data_init

```

Replace occurrences of GNU path variables (such as ``${prefix}``) by their values. Do this for all strings that could depend on them, and do the replacement in reverse order, since the path variables may be defined in terms of each other.

(*OS interface: procedures*)+≡

```

subroutine os_data_expand_paths (os_data)
    type(os_data_t), intent(inout) :: os_data
    integer, parameter :: N_VARIABLES = 6
    type(string_t), dimension(N_VARIABLES) :: variable, value
    variable(1) = "${prefix}";      value(1) = os_data%prefix
    variable(2) = "${exec_prefix}"; value(2) = os_data%exec_prefix
    variable(3) = "${bindir}";      value(3) = os_data%bindir
    variable(4) = "${libdir}";      value(4) = os_data%libdir
    variable(5) = "${includedir}";  value(5) = os_data%includedir
    variable(6) = "${datarootdir}"; value(6) = os_data%datarootdir
    call expand_paths (os_data%whizard_omega_binpath)
    call expand_paths (os_data%whizard_includes)
    call expand_paths (os_data%whizard_ldflags)
    call expand_paths (os_data%whizard_libtool)
    call expand_paths (os_data%whizard_modelpath)
    call expand_paths (os_data%whizard_models_libpath)
    call expand_paths (os_data%whizard_susypath)
    call expand_paths (os_data%whizard_gmlpath)
    call expand_paths (os_data%whizard_cutspath)
    call expand_paths (os_data%whizard_texpath)
    call expand_paths (os_data%whizard_testdatapath)
    call expand_paths (os_data%whizard_circe2path)
    call expand_paths (os_data%whizard_beamsimpath)

```

```

call expand_paths (os_data%whizard_mulipath)
call expand_paths (os_data%whizard_models_libpath_local)
call expand_paths (os_data%whizard_modelpath_local)
call expand_paths (os_data%whizard_omega_binpath_local)
call expand_paths (os_data%pdf_builtin_datapath)
call expand_paths (os_data%latex)
call expand_paths (os_data%mpost)
call expand_paths (os_data%gml)
call expand_paths (os_data%dvips)
call expand_paths (os_data%ps2pdf)
contains
subroutine expand_paths (string)
  type(string_t), intent(inout) :: string
  integer :: i
  do i = N_VARIABLES, 1, -1
    string = replace (string, variable(i), value(i), every=.true.)
  end do
end subroutine expand_paths
end subroutine os_data_expand_paths

```

Write contents

*<OS interface: public>+≡*

```
public :: os_data_write
```

*<OS interface: procedures>+≡*

```

subroutine os_data_write (os_data, unit)
  type(os_data_t), intent(in) :: os_data
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(A)") "OS data:"
  write (u, *) "use_libtool    = ", os_data%use_libtool
  write (u, *) "use_testfiles  = ", os_data%use_testfiles
  write (u, *) "fc            = ", char (os_data%fc)
  write (u, *) "fcflags       = ", char (os_data%fcflags)
  write (u, *) "fcflags_pic    = ", char (os_data%fcflags_pic)
  write (u, *) "fc_src_ext     = ", char (os_data%fc_src_ext)
  write (u, *) "cc            = ", char (os_data%cc)
  write (u, *) "cflags        = ", char (os_data%cflags)
  write (u, *) "cflags_pic    = ", char (os_data%cflags_pic)
  write (u, *) "obj_ext       = ", char (os_data%obj_ext)
  write (u, *) "ld            = ", char (os_data%ld)
  write (u, *) "ldflags       = ", char (os_data%ldflags)
  write (u, *) "ldflags_so     = ", char (os_data%ldflags_so)
  write (u, *) "ldflags_static = ", char (os_data%ldflags_static)
  write (u, *) "ldflags_hepmc  = ", char (os_data%ldflags_hepmc)
  write (u, *) "shlib_ext     = ", char (os_data%shlib_ext)
  write (u, *) "makeflags     = ", char (os_data%makeflags)
  write (u, *) "prefix        = ", char (os_data%prefix)
  write (u, *) "exec_prefix   = ", char (os_data%exec_prefix)
  write (u, *) "bindir       = ", char (os_data%bindir)
  write (u, *) "libdir       = ", char (os_data%libdir)
  write (u, *) "includedir   = ", char (os_data%includedir)
  write (u, *) "datarootdir  = ", char (os_data%datarootdir)

```

```

write (u, *) "whizard_omega_binpath = ", &
char (os_data%whizard_omega_binpath)
write (u, *) "whizard_includes      = ", char (os_data%whizard_includes)
write (u, *) "whizard_ldflags      = ", char (os_data%whizard_ldflags)
write (u, *) "whizard_libtool      = ", char (os_data%whizard_libtool)
write (u, *) "whizard_modelpath    = ", &
char (os_data%whizard_modelpath)
write (u, *) "whizard_models_libpath = ", &
char (os_data%whizard_modelpath)
write (u, *) "whizard_susypath      = ", char (os_data%whizard_susypath)
write (u, *) "whizard_gmlpath      = ", char (os_data%whizard_gmlpath)
write (u, *) "whizard_cutspath     = ", char (os_data%whizard_cutspath)
write (u, *) "whizard_texpath      = ", char (os_data%whizard_texpath)
write (u, *) "whizard_circe2path    = ", char (os_data%whizard_circe2path)
write (u, *) "whizard_beamsimpath   = ", char (os_data%whizard_beamsimpath)
write (u, *) "whizard_mulipath     = ", char (os_data%whizard_mulipath)
write (u, *) "whizard_testdatapath = ", &
char (os_data%whizard_testdatapath)
write (u, *) "whizard_modelpath_local = ", &
char (os_data%whizard_modelpath_local)
write (u, *) "whizard_models_libpath_local = ", &
char (os_data%whizard_models_libpath_local)
write (u, *) "whizard_omega_binpath_local = ", &
char (os_data%whizard_omega_binpath_local)
write (u, *) "event_analysis      = ", os_data%event_analysis
write (u, *) "event_analysis_ps    = ", os_data%event_analysis_ps
write (u, *) "event_analysis_pdf   = ", os_data%event_analysis_pdf
write (u, *) "latex              = ", char (os_data%latex)
write (u, *) "mpost              = ", char (os_data%mpost)
write (u, *) "gml              = ", char (os_data%gml)
write (u, *) "dvips              = ", char (os_data%dvips)
write (u, *) "ps2pdf             = ", char (os_data%ps2pdf)
end subroutine os_data_write

```

### 2.5.3 Dynamic linking

We define a type that holds the filehandle for a dynamically linked library (shared object), together with functions to open and close the library, and to access functions in this library.

```

<OS interface: public>+≡
public :: dlaccess_t

<OS interface: types>+≡
type :: dlaccess_t
private
type(string_t) :: filename
type(c_ptr) :: handle = c_null_ptr
logical :: is_open = .false.
logical :: has_error = .false.
type(string_t) :: error
contains
<OS interface: dlaccess: TBP>
end type dlaccess_t

```

Output. This is called by the output routine for the process library.

```

<OS interface: dlaccess: TBP>≡
  procedure :: write => dlaccess_write
<OS interface: procedures>+≡
  subroutine dlaccess_write (object, unit)
    class(dlaccess_t), intent(in) :: object
    integer, intent(in) :: unit
    write (unit, "(1x,A)") "DL access info:"
    !!! This is OS dependent (could be wrapped with a logical though)
    !!! write (unit, "(3x,A,A,A)") "filename = ', char (object%filename), ""
    write (unit, "(3x,A,L1)") "is open = ", object%is_open
    if (object%has_error) then
      write (unit, "(3x,A,A,A)") "error = ', char (object%error), ""
    else
      write (unit, "(3x,A)") "error = [none]"
    end if
  end subroutine dlaccess_write

```

The interface to the library functions:

```

<OS interface: interfaces>≡
  interface
    function dlopen (filename, flag) result (handle) bind(C)
      import
      character(c_char), dimension(*) :: filename
      integer(c_int), value :: flag
      type(c_ptr) :: handle
    end function dlopen
  end interface

  interface
    function dlclose (handle) result (status) bind(C)
      import
      type(c_ptr), value :: handle
      integer(c_int) :: status
    end function dlclose
  end interface

  interface
    function dlerror () result (str) bind(C)
      import
      type(c_ptr) :: str
    end function dlerror
  end interface

  interface
    function dlsym (handle, symbol) result (fptr) bind(C)
      import
      type(c_ptr), value :: handle
      character(c_char), dimension(*) :: symbol
      type(c_funptr) :: fptr
    end function dlsym
  end interface

```



```
end interface
```

This reads an error string and transforms it into a `string_t` object, if an error has occurred. If not, set the error flag to false and return an empty string.

```
<Limits: public parameters>+≡
    integer, parameter, public :: DLERROR_LEN = 160

<OS interface: procedures>+≡
    subroutine read_dlerror (has_error, error)
        logical, intent(out) :: has_error
        type(string_t), intent(out) :: error
        type(c_ptr) :: err_cptr
        character(len=DLERROR_LEN, kind=c_char), pointer :: err_fptr
        integer :: str_end
        err_cptr = dlerror ()
        if (c_associated (err_cptr)) then
            call c_f_pointer (err_cptr, err_fptr)
            has_error = .true.
            str_end = scan (err_fptr, c_null_char)
            if (str_end > 0) then
                error = err_fptr(1:str_end-1)
            else
                error = err_fptr
            end if
        else
            has_error = .false.
            error = ""
        end if
    end subroutine read_dlerror
```

This is the Fortran API. Init/final open and close the file, i.e., load and unload the library.

Note that a library can be opened more than once, and that for an ultimate close as many `dlclose` calls as `dlopen` calls are necessary. However, we assume that it is opened and closed only once.

```
<OS interface: public>+≡
    public :: dlaccess_init
    public :: dlaccess_final

<OS interface: dlaccess: TBP>+≡
    procedure :: init => dlaccess_init
    procedure :: final => dlaccess_final

<OS interface: procedures>+≡
    subroutine dlaccess_init (dlaccess, prefix, libname, os_data)
        class(dlaccess_t), intent(out) :: dlaccess
        type(string_t), intent(in) :: prefix, libname
        type(os_data_t), intent(in), optional :: os_data
        type(string_t) :: filename
        logical :: exist
        dlaccess%filename = libname
        filename = prefix // "/" // libname
        inquire (file=char(filename), exist=exist)
        if (.not. exist) then
```

```

        filename = prefix // "/.libs/" // libname
        inquire (file=char(filename), exist=exist)
        if (.not. exist) then
            dlaccess%has_error = .true.
            dlaccess%error = "Library '" // filename // "' not found"
            return
        end if
    end if
    dlaccess%handle = dlopen (char (filename) // c_null_char, ior ( &
        RTLD_LAZY, RTLD_LOCAL))
    dlaccess%is_open = c_associated (dlaccess%handle)
    call read_dlerror (dlaccess%has_error, dlaccess%error)
end subroutine dlaccess_init

subroutine dlaccess_final (dlaccess)
    class(dlaccess_t), intent(inout) :: dlaccess
    integer(c_int) :: status
    if (dlaccess%is_open) then
        status = dlclose (dlaccess%handle)
        dlaccess%is_open = .false.
        call read_dlerror (dlaccess%has_error, dlaccess%error)
    end if
end subroutine dlaccess_final

```

Return true if an error has occurred.

```

<OS interface: public>+≡
    public :: dlaccess_has_error

<OS interface: procedures>+≡
    function dlaccess_has_error (dlaccess) result (flag)
        logical :: flag
        type(dlaccess_t), intent(in) :: dlaccess
        flag = dlaccess%has_error
    end function dlaccess_has_error

```

Return the error string currently stored in the dlaccess object.

```

<OS interface: public>+≡
    public :: dlaccess_get_error

<OS interface: procedures>+≡
    function dlaccess_get_error (dlaccess) result (error)
        type(string_t) :: error
        type(dlaccess_t), intent(in) :: dlaccess
        error = dlaccess%error
    end function dlaccess_get_error

```

The symbol handler returns the C address of the function with the given string name. (It is a good idea to use `bind(C)` for all functions accessed by this, such that the name string is well-defined.) Call `c_f_procpointer` to cast this into a Fortran procedure pointer with an appropriate interface.

```

<OS interface: public>+≡
    public :: dlaccess_get_c_funptr

```

```

<OS interface: procedures>+≡
function dlaccess_get_c_funptr (dlaccess, fname) result (fptr)
  type(c_funptr) :: fptr
  type(dlaccess_t), intent(inout) :: dlaccess
  type(string_t), intent(in) :: fname
  fptr = dlsym (dlaccess%handle, char (fname) // c_null_char)
  call read_dlerror (dlaccess%has_error, dlaccess%error)
end function dlaccess_get_c_funptr

```

## 2.5.4 Predicates

Return true if the library is loaded. In particular, this is false if loading was unsuccessful.

```

<OS interface: public>+≡
  public :: dlaccess_is_open

<OS interface: procedures>+≡
function dlaccess_is_open (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%is_open
end function dlaccess_is_open

```

## 2.5.5 Shell access

This is the standard system call for executing a shell command, such as invoking a compiler.

In F2008 there will be the equivalent built-in command `execute_command_line`.

```

<OS interface: public>+≡
  public :: os_system_call

<OS interface: procedures>+≡
subroutine os_system_call (command_string, status, verbose)
  type(string_t), intent(in) :: command_string
  integer, intent(out), optional :: status
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: stat
  verb = .false.; if (present (verbose)) verb = verbose
  if (verb) &
    call msg_message ("command: " // char (command_string))
  stat = system (char (command_string) // c_null_char)
  if (present (status)) then
    status = stat
  else if (stat /= 0) then
    if (.not. verb) &
      call msg_message ("command: " // char (command_string))
    write (msg_buffer, "(A,I0)") "Return code = ", stat
    call msg_message ()
    call msg_fatal ("System command returned with nonzero status code")
  end if

```

```

end subroutine os_system_call

<OS interface: interfaces>+≡
interface
  function system (command) result (status) bind(C)
    import
    integer(c_int) :: status
    character(c_char), dimension(*) :: command
  end function system
end interface

```

## 2.5.6 Querying for a directory

This queries for the existence of a directory. There is no standard way to achieve this in FORTRAN, and if we were to call into `libc`, we would need access to C macros for evaluating the result, so we resort to calling `test` as a system call.

```

<OS interface: public>+≡
public :: os_dir_exist

<OS interface: procedures>+≡
function os_dir_exist (name) result (res)
  type(string_t), intent(in) :: name
  logical :: res
  integer :: status
  call os_system_call ('test -d "' // name // '"', status=status)
  res = status == 0
end function os_dir_exist

```

## 2.5.7 Fortran compiler and linker

Compile a single module for use in a shared library, but without linking.

```

<OS interface: public>+≡
public :: os_compile_shared

<OS interface: procedures>+≡
subroutine os_compile_shared (src, os_data, status)
  type(string_t), intent(in) :: src
  type(os_data_t), intent(in) :: os_data
  integer, intent(out), optional :: status
  type(string_t) :: command_string
  if (os_data%use_libtool) then
    command_string = &
      os_data%whizard_libtool // " --mode=compile " // &
      os_data%fc // " " // &
      "-c " // &
      os_data%whizard_includes // " " // &
      os_data%fcflags // " " // &
      "' " // src // os_data%fc_src_ext // "' "
  else
    command_string = &
      os_data%fc // " " // &
      "-c " // &

```

```

        os_data%fcflags_pic // " " // &
        os_data%whizard_includes // " " // &
        os_data%fcflags // " " // &
        "" // src // os_data%fc_src_ext // ""
    end if
    call os_system_call (command_string, status)
end subroutine os_compile_shared

```

Link an array of object files to build a shared object library. In the libtool case, we have to specify a `-rpath`, otherwise only a static library can be built. However, since the library is never installed, this `rpath` is irrelevant.

```

<OS interface: public>+≡
    public :: os_link_shared

<OS interface: procedures>+≡
    subroutine os_link_shared (objlist, lib, os_data, status)
        type(string_t), intent(in) :: objlist, lib
        type(os_data_t), intent(in) :: os_data
        integer, intent(out), optional :: status
        type(string_t) :: command_string
        if (os_data%use_libtool) then
            command_string = &
                os_data%whizard_libtool // " --mode=link " // &
                os_data%fc // " " // &
                "-module " // &
                "-rpath /usr/local/lib" // " " // &
                os_data%fcflags // " " // &
                os_data%whizard_ldflags // " " // &
                os_data%ldflags // " " // &
                "-o '" // lib // ".la' " // &
                objlist
        else
            command_string = &
                os_data%ld // " " // &
                os_data%ldflags_so // " " // &
                os_data%fcflags // " " // &
                os_data%whizard_ldflags // " " // &
                os_data%ldflags // " " // &
                "-o '" // lib // os_data%shlib_ext // "' " // &
                objlist
        end if
        call os_system_call (command_string, status)
    end subroutine os_link_shared

```

Link an array of object files / libraries to build a static executable.

```

<OS interface: public>+≡
    public :: os_link_static

<OS interface: procedures>+≡
    subroutine os_link_static (objlist, exec_name, os_data, status)
        type(string_t), intent(in) :: objlist, exec_name
        type(os_data_t), intent(in) :: os_data
        integer, intent(out), optional :: status
        type(string_t) :: command_string

```

```

if (os_data%use_libtool) then
  command_string = &
    os_data%whizard_libtool // " --mode=link " // &
    os_data%fc // " " // &
    "-static-libtool-libs " // &
    os_data%fcflags // " " // &
    os_data%whizard_ldflags // " " // &
    os_data%ldflags // " " // &
    os_data%ldflags_static // " " // &
    "-o '" // exec_name // "' " // &
    objlist // " " // &
    os_data%ldflags_hepmc
else
  command_string = &
    os_data%ld // " " // &
    os_data%ldflags_so // " " // &
    os_data%fcflags // " " // &
    os_data%whizard_ldflags // " " // &
    os_data%ldflags // " " // &
    os_data%ldflags_static // " " // &
    "-o '" // exec_name // "' " // &
    objlist // " " // &
    os_data%ldflags_hepmc
end if
call os_system_call (command_string, status)
end subroutine os_link_static

```

Determine the name of the shared library to link. If libtool is used, this is encoded in the .la file which resides in place of the library itself.

```

<OS interface: public>+≡
  public :: os_get_dlname

<OS interface: procedures>+≡
  function os_get_dlname (lib, os_data, ignore, silent) result (dlname)
    type(string_t) :: dlname
    type(string_t), intent(in) :: lib
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: ignore, silent
    type(string_t) :: filename
    type(string_t) :: buffer
    logical :: exist, required, quiet
    integer :: u
    u = free_unit ()
    if (present (ignore)) then
      required = .not. ignore
    else
      required = .true.
    end if
    if (present (silent)) then
      quiet = silent
    else
      quiet = .false.
    end if
    if (os_data%use_libtool) then

```

```

filename = lib // ".la"
inquire (file=char(filename), exist=exist)
if (exist) then
  open (unit=u, file=char(filename), action="read", status="old")
  SCAN_LTFILE: do
    call get (u, buffer)
    if (extract (buffer, 1, 7) == "dlname=") then
      dlname = extract (buffer, 9)
      dlname = remove (dlname, len (dlname))
      exit SCAN_LTFILE
    end if
  end do SCAN_LTFILE
  close (u)
else if (required) then
  if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
    // "': libtool archive not found")
  dlname = ""
else
  if (.not. quiet) call msg_message ("[No compiled library '" &
    // char (lib) // "']")
  dlname = ""
end if
else
  dlname = lib // os_data%shlib_ext
  inquire (file=char(dlname), exist=exist)
  if (.not. exist) then
    if (required) then
      if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
        // "' not found")
    else
      if (.not. quiet) call msg_message &
        ("[No compiled process library '" // char (lib) // "']")
      dlname = ""
    end if
  end if
end if
end if
end function os_get_dlname

```

## 2.5.8 Controlling OpenMP

OpenMP is handled automatically by the library for the most part. Here is a convenience routine for setting the number of threads, with some diagnostics.

```

<OS interface: public>+≡
  public :: openmp_set_num_threads_verbose

<OS interface: procedures>+≡
  subroutine openmp_set_num_threads_verbose (num_threads)
    integer, intent(in) :: num_threads
    integer :: n_threads
    n_threads = num_threads
    if (openmp_is_active ()) then
      if (num_threads == 1) then
        write (msg_buffer, "(A,I0,A)") "OpenMP: Using ", num_threads, &

```

```

        " thread"
        call msg_message
        n_threads = num_threads
    else if (num_threads > 1) then
        write (msg_buffer, "(A,I0,A)") "OpenMP: Using ", num_threads, &
            " threads"
        call msg_message
        n_threads = num_threads
    else
        write (msg_buffer, "(A,I0,A)") "OpenMP: " &
            // "Illegal value of openmp_num_threads (", num_threads, &
            ") ignored"
        call msg_error
        n_threads = openmp_get_default_max_threads ()
        write (msg_buffer, "(A,I0,A)") "OpenMP: Using ", &
            n_threads, " threads"
        call msg_message
    end if
    if (n_threads > openmp_get_default_max_threads ()) then
        write (msg_buffer, "(A,I0)") "OpenMP: " &
            // "Number of threads is greater than library default of ", &
            openmp_get_default_max_threads ()
        call msg_warning
    end if
    call openmp_set_num_threads (n_threads)
    else if (num_threads /= 1) then
        write (msg_buffer, "(A,I0,A)") "openmp_num_threads set to ", &
            num_threads, ", but OpenMP is not active: ignored"
        call msg_warning
    end if
end if
end subroutine openmp_set_num_threads_verbose

```

### 2.5.9 Test

```

<OS interface: public>+≡
    public :: os_interface_test

<OS interface: procedures>+≡
    subroutine os_interface_test ()
        call os_interface_test1 ()
    end subroutine os_interface_test

```

Write a Fortran source file, compile it to a shared library, load it, and execute the contained function.

```

<OS interface: procedures>+≡
    subroutine os_interface_test1 ()
        use diagnostics, only: msg_fatal !NODEP!
        type(dlaccess_t) :: dlaccess
        type(string_t) :: fname, libname, ext
        type(os_data_t) :: os_data
        type(string_t) :: filename_src, filename_obj
        abstract interface

```



```

        function so_test_proc (i) result (j) bind(C)
            import c_int
            integer(c_int), intent(in) :: i
            integer(c_int) :: j
        end function so_test_proc
end interface
procedure(so_test_proc), pointer :: so_test => null ()
type(c_funptr) :: c_fptr
integer :: u
integer(c_int) :: i
call os_data_init (os_data)
fname = "so_test"
filename_src = fname // os_data%fc_src_ext
if (os_data%use_libtool) then
    ext = ".lo"
else
    ext = os_data%obj_ext
end if
filename_obj = fname // ext
libname = fname // os_data%shlib_ext
print *, "* write source file 'so_test.f90'"
u = free_unit ()
open (unit=u, file=char(filename_src), action="write")
write (u, "(A)") "function so_test (i) result (j) bind(C)"
write (u, "(A)") "  use iso_c_binding"
write (u, "(A)") "  integer(c_int), intent(in) :: i"
write (u, "(A)") "  integer(c_int) :: j"
write (u, "(A)") "  j = 2 * i"
write (u, "(A)") "end function so_test"
close (u)
print *, "* compile and link as 'so_test.so'"
call os_compile_shared (fname, os_data)
call os_link_shared (filename_obj, fname, os_data)
print *, "* load library 'so_test.so'"
call dlaccess_init (dlaccess, var_str ("."), libname, os_data)
if (dlaccess_is_open (dlaccess)) then
    print *, "  success"
else
    print *, "  failure"
end if
print *, "* load symbol 'so_test'"
c_fptr = dlaccess_get_c_funptr (dlaccess, fname)
if (c_associated (c_fptr)) then
    print *, "  success"
else
    print *, "  failure"
end if
call c_f_procpointer (c_fptr, so_test)
print *, "* Execute function from 'so_test.so'"
i = 7
print *, "  input = ", i
print *, "  result =", so_test(i)
if (so_test(i) / i .ne. 2) then
    call msg_fatal ("Compiling and linking ISO C functions failed.")

```

```

    else
        print *, "* Successful."
    end if
    print *, "* Cleanup"
    call dlaccess_final (dlaccess)
end subroutine os_interface_test1

```

## 2.6 CPU timing

This is a simplified module which replaces `clock.f90` below. The time is now stored in a simple derived type which just holds a floating-point number.

*<cputime.f90>≡*  
*<File header>*

```

module cputime

```

*<Use kinds>*

*<Use file utils>*

*<Use strings>*

```

use diagnostics !NODEP!

```

*<Standard module head>*

*<CPU time: public>*

*<CPU time: types>*

*<CPU time: interfaces>*

```

contains

```

*<CPU time: procedures>*

```

end module cputime

```

The CPU time is a floating-point number with an arbitrary reference time. It is single precision (default real, not `real(default)`). It is measured in seconds.

*<CPU time: public>≡*

```

    public :: time_t

```

*<CPU time: types>≡*

```

    type :: time_t

```

```

        private

```

```

        logical :: known = .false.

```

```

        real :: value = 0

```

```

    end type time_t

```

*<CPU time: public>+≡*

```

    public :: time_write

```

*<CPU time: procedures>≡*

```

    subroutine time_write (time, unit)

```

```

        type(time_t), intent(in) :: time

```

```

integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit)
write (u, "(A)", advance="no") "Time in seconds = "
if (time%known) then
    write (u, *) time%value
else
    write (u, *) "[unknown]"
end if
end subroutine time_write

```

Set the current time

```

<CPU time: public>+≡
public :: time_current
<CPU time: procedures>+≡
function time_current () result (time)
    type(time_t) :: time
    integer :: msecs
    call system_clock (msecs)
    time%value = real (msecs) / 1000.
    time%known = time%value > 0
end function time_current

```

Assign to a real(default value. If the time is undefined, return zero.

```

<CPU time: public>+≡
public :: assignment(=)
<CPU time: interfaces>≡
interface assignment(=)
    module procedure real_assign_time
end interface
<CPU time: procedures>+≡
pure subroutine real_assign_time (r, time)
    real(default), intent(out) :: r
    type(time_t), intent(in) :: time
    if (time%known) then
        r = time%value
    else
        r = 0
    end if
end subroutine real_assign_time

```

Only time differences have a real meaning. If any input value is undefined, the result is undefined.

```

<CPU time: public>+≡
public :: operator(-)
<CPU time: interfaces>+≡
interface operator(-)
    module procedure subtract_times
end interface

```

```

<CPU time: procedures>+≡
pure function subtract_times (t_end, t_begin) result (time)
  type(time_t) :: time
  type(time_t), intent(in) :: t_end, t_begin
  if (t_end%known .and. t_begin%known) then
    time%known = .true.
    time%value = t_end%value - t_begin%value
  end if
end function subtract_times

```

We define functions for converting the time into ss / mm:ss / hh:mm:ss / dd:mm:hh:ss.

```

<CPU time: public>+≡
public :: time_retrieve
public :: time2string_s
public :: time2string_ms
public :: time2string_hms
public :: time2string_dhms
public :: time2string

<CPU time: interfaces>+≡
interface time_retrieve
  module procedure time_retrieve_s, time_retrieve_ms, time_retrieve_hms, &
    time_retrieve_dhms
end interface time_retrieve

<CPU time: procedures>+≡
subroutine time_retrieve_s (time, sec)
  integer, intent(in) :: time
  integer, intent(out) :: sec
  sec = time
end subroutine time_retrieve_s

subroutine time_retrieve_ms (time, min, sec)
  integer, intent(in) :: time
  integer, intent(out) :: min, sec
  sec = mod (time, 60)
  min = time / 60
end subroutine time_retrieve_ms

subroutine time_retrieve_hms (time, hour, min, sec)
  integer, intent(in) :: time
  integer, intent(out) :: hour, min, sec
  call time_retrieve_ms (time, min, sec)
  hour = min / 60
  min = mod (min, 60)
end subroutine time_retrieve_hms

subroutine time_retrieve_dhms (time, day, hour, min, sec)
  integer, intent(in) :: time
  integer, intent(out) :: day, hour, min, sec
  call time_retrieve_hms (time, hour, min, sec)
  day = hour / 24
  hour = mod (hour, 24)
end subroutine time_retrieve_dhms

```

```

function time2string_s (time) result (str)
    integer, intent(in) :: time
    type(string_t) :: str
    str = int2string (time) // "s"
end function time2string_s

function time2string_ms (time) result (str)
    integer, intent(in) :: time
    type(string_t) :: str
    integer :: min, sec
    call time_retrieve (time, min, sec)
    str = int2string (min) // "m:" // int2string (sec) // "s"
end function time2string_ms

function time2string_hms (time) result (str)
    integer, intent(in) :: time
    type(string_t) :: str
    integer :: hour, min, sec
    call time_retrieve (time, hour, min, sec)
    str = int2string (hour) // "h:" // &
        int2string (min) // "m:" // int2string (sec) // "s"
end function time2string_hms

function time2string_dhms (time) result (str)
    integer, intent(in) :: time
    type(string_t) :: str
    integer :: day, hour, min, sec
    call time_retrieve (time, day, hour, min, sec)
    str = int2string (day) // "d:" // int2string (hour) // "h:" // &
        int2string (min) // "m:" // int2string (sec) // "s"
end function time2string_dhms

function time2string (time) result (str)
    integer, intent(in) :: time
    type(string_t) :: str
    integer :: day, hour, min, sec
    call time_retrieve (time, day, hour, min, sec)
    if (day /= 0) then
        str = time2string_dhms (time)
    else if (hour /= 0) then
        str = time2string_hms (time)
    else if (min /= 0) then
        str = time2string_ms (time)
    else
        str = time2string_s (time)
    end if
end function time2string

```

## 2.7 Accessing the system clock

*This module is currently disabled, we are using the module above instead.*

Fortran 90 provides a standard interface for the system clock. Here, we define a record for storing the information, and higher-level functions for accessing it.

```

<clock.f90>≡
  <File header>

  module clock

    <Use kinds>
    <Use file utils>

    <Standard module head>

    <Clock: public>

    <Clock: types>

    <Clock: interfaces>

    contains

    <Clock: procedures>

  end module clock

```

The format for storing the current date. We want to be able to store time differences, therefore everything is converted to the time difference between now and Jan 01 2000, 0:00. The timezone is ignored.

```

<Clock: public>≡
  public :: time_t

<Clock: types>≡
  type :: time_t
    integer :: day = 0
    integer :: hour = 0
    integer :: minute = 0
    integer :: second = 0
    integer :: millisecond = 0
  end type time_t

```

Write the time in readable form

```

<Clock: public>+≡
  public :: time_write

<Clock: interfaces>≡
  interface time_write
    module procedure time_write_unit
    module procedure time_write_string
  end interface

```

The maximal time that can be written is 999 days. No way to write negative times.

```

<Clock: procedures>≡
  subroutine time_write_unit (t, unit, advance, days, seconds, milliseconds)
    type(time_t), intent(in) :: t
    integer, intent(in) :: unit
    character(len=*), intent(in), optional :: advance

```

```

logical, intent(in), optional :: days, seconds, milliseconds
logical :: dd, ss, ms
character(len=3) :: adv
if (present (advance)) then
    adv = advance
else
    adv = "yes"
end if
dd = .false.; if (present (days)) dd = days
ss = .true.; if (present (seconds)) ss = seconds
ms = .false.; if (present (milliseconds)) ms = milliseconds
if (dd) then
    write (unit, "(I3,A,1x,I2.2,A,1x,I2.2,A)", advance="no") &
        t%day, "d", t%hour, "h", t%minute, "m"
else
    write (unit, "(I5,A,1x,I2.2,A)", advance="no") &
        t%day * 24 + t%hour, "h", t%minute, "m"
end if
if (ss) then
    write (unit, "(1x,I2.2)", advance="no") t%second
    if (ms) then
        write (unit, "(A,I3.3,A)", advance=trim(adv)) ".", t%millisecond, "s"
    else
        write (unit, "(A)", advance=trim(adv)) "s"
    end if
end if
end subroutine time_write_unit

```

Write the time to a string. This uses a scratch file for simplicity. Non-advancing output is not possible.

```

<Clock: procedures>+≡
subroutine time_write_string (t, string, days, seconds, milliseconds)
    type(time_t), intent(in) :: t
    character(*), intent(out) :: string
    logical, intent(in), optional :: days, seconds, milliseconds
    integer :: unit
    unit = free_unit ()
    open (unit, status="scratch", action="readwrite")
    call time_write_unit (t, unit, &
        days=days, seconds=seconds, milliseconds=milliseconds)
    rewind (unit)
    read (unit, "(A)") string
    close (unit)
end subroutine time_write_string

```

Set the current time to number of days, minutes, seconds, milliseconds elapsed since Jan 1 2000. Works for dates after Jan 1 2001 until Dec 31 2099.

```

<Clock: public>+≡
public :: time_current

<Clock: procedures>+≡
function time_current () result (t)
    type(time_t) :: t

```

```

integer, dimension(8) :: values
integer :: year
call date_and_time (values = values)
t%millisecond = values(8)
t%second = values(7)
t%minute = values(6)
t%hour = values(5)
! values(4) is the difference local time - GMT in minutes
t%day = values(3) - 1
select case (values(2))
case ( 1)
case ( 2); t%day = t%day + 31
case ( 3); t%day = t%day + 31 + 28
case ( 4); t%day = t%day + 31 + 28 + 31
case ( 5); t%day = t%day + 31 + 28 + 31 + 30
case ( 6); t%day = t%day + 31 + 28 + 31 + 30 + 31
case ( 7); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30
case ( 8); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31
case ( 9); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31
case (10); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30
case (11); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31
case (12); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30
end select
year = values(1) - 2000
if (year < 1) return
t%day = t%day + 365 * year + ((year-1) / 4 + 1)
if (mod (year, 4) == 0 .and. values(2) > 2) t%day = t%day + 1
end function time_current

```

Convert from and to seconds

```

<Clock: public>+≡
public :: time_to_seconds, time_from_seconds

<Clock: procedures>+≡
function time_to_seconds (t) result (s)
type(time_t), intent(in) :: t
real(kind=default) :: s
s = t%millisecond / 1000._default &
+ t%second &
+ 60._default * (t%minute &
+ 60._default * (t%hour &
+ 24._default * t%day))
end function time_to_seconds

<Clock: procedures>+≡
function time_from_seconds (s) result (t)
real(kind=default), intent(in) :: s
type(time_t) :: t
t%millisecond = s * 1000
t%second = t%millisecond / 1000
t%millisecond = mod (t%millisecond, 1000)
t%minute = t%second / 60
t%second = mod (t%second, 60)
t%hour = t%minute / 60

```



```

    t%minute = mod (t%minute, 60)
    t%day = t%hour / 24
    t%hour = mod (t%hour, 24)
end function time_from_seconds

```

Add and subtract times

```

<Clock: public>+≡
    public :: operator(+), operator(-)

<Clock: interfaces>+≡
    interface operator(+)
        module procedure time_add
    end interface
    interface operator(-)
        module procedure time_subtract
    end interface

<Clock: procedures>+≡
    function time_add (t1, t2) result (t)
        type(time_t), intent(in) :: t1, t2
        type(time_t) :: t
        t%millisecond = t1%millisecond + t2%millisecond
        t%second = t1%second + t2%second
        t%minute = t1%minute + t2%minute
        t%hour = t1%hour + t2%hour
        t%day = t1%day + t2%day
        if (t%millisecond > 999) then
            t%second = t%second + t%millisecond / 1000
            t%millisecond = modulo (t%millisecond, 1000)
        end if
        if (t%second > 59) then
            t%minute = t%minute + t%second / 60
            t%second = modulo (t%second, 60)
        end if
        if (t%minute > 59) then
            t%hour = t%hour + t%minute / 60
            t%minute = modulo (t%minute, 60)
        end if
        if (t%hour > 23) then
            t%day = t%day + t%hour / 24
            t%hour = modulo (t%hour, 24)
        end if
    end function time_add

```

This works for positive difference only, the first time must be later than the second. Otherwise, the number of days is set to zero.

```

<Clock: procedures>+≡
    function time_subtract (t1, t2) result (t)
        type(time_t), intent(in) :: t1, t2
        type(time_t) :: t
        t%millisecond = t1%millisecond - t2%millisecond
        t%second = t1%second - t2%second
        t%minute = t1%minute - t2%minute
        t%hour = t1%hour - t2%hour

```

```

t%day = t1%day - t2%day
if (t%millisecond < 0) then
    t%second = t%second - 1 + t%millisecond / 1000
    t%millisecond = modulo (t%millisecond, 1000)
end if
if (t%second < 0) then
    t%minute = t%minute - 1 + t%second / 60
    t%second = modulo (t%second, 60)
end if
if (t%minute < 0) then
    t%hour = t%hour - 1 + t%minute / 60
    t%minute = modulo (t%minute, 60)
end if
if (t%hour < 0) then
    t%day = t%day - 1 + t%hour / 24
    t%hour = modulo (t%hour, 24)
end if
if (t%day < 0) then
    t%day = 0
end if
end function time_subtract

```

Compare times

*<Clock: public>+≡*

```

public :: operator(==)
public :: operator(<), operator(>)
public :: operator(<=), operator(>=)

```

*<Clock: interfaces>+≡*

```

interface operator(==)
    module procedure time_equal
end interface
interface operator(<)
    module procedure time_less_than
end interface
interface operator(>)
    module procedure time_greater_than
end interface
interface operator(<=)
    module procedure time_less_equal
end interface
interface operator(>=)
    module procedure time_greater_equal
end interface

```

*<Clock: procedures>+≡*

```

function time_equal (t1, t2) result (equal)
    type(time_t), intent(in) :: t1, t2
    logical :: equal
    equal = (t1%day==t2%day) .and. (t1%hour==t2%hour) .and. &
        (t1%minute==t2%minute) .and. (t1%second==t2%second) .and. &
        (t1%millisecond==t2%millisecond)
end function time_equal

```

```

<Clock: procedures>+≡
function time_less_than (t1, t2) result (less)
  type(time_t), intent(in) :: t1, t2
  logical :: less
  if (t1%day < t2%day) then
    less = .true.
  else if (t1%day == t2%day) then
    if (t1%hour < t2%hour) then
      less = .true.
    else if (t1%hour == t2%hour) then
      if (t1%minute < t2%minute) then
        less = .true.
      else if (t1%minute == t2%minute) then
        if (t1%second < t2%second) then
          less = .true.
        else if (t1%second == t2%second) then
          if (t1%millisecond < t2%millisecond) then
            less = .true.
          else
            less = .false.
          end if
        else
          less = .false.
        end if
      else
        less = .false.
      end if
    else
      less = .false.
    end if
  else
    less = .false.
  end if
end function time_less_than

```

```

<Clock: procedures>+≡
function time_greater_than (t1, t2) result (greater)
  type(time_t), intent(in) :: t1, t2
  logical :: greater
  greater = time_less_than (t2, t1)
end function time_greater_than

```

```

<Clock: procedures>+≡
function time_less_equal (t1, t2) result (less_equal)
  type(time_t), intent(in) :: t1, t2
  logical :: less_equal
  less_equal = time_equal (t1, t2) .or. time_less_than (t1, t2)
end function time_less_equal

```

```

<Clock: procedures>+≡
function time_greater_equal (t1, t2) result (greater_equal)
  type(time_t), intent(in) :: t1, t2
  logical :: greater_equal

```

```

    greater_equal = time_equal (t1, t2) .or. time_greater_than (t1, t2)
end function time_greater_equal

```

## 2.8 Hashtables

Hash tables, like lists, are not part of Fortran and must be defined on a per-case basis. In this section we define a module that contains a hash function.

Furthermore, for reference there is a complete framework of hashtable type definitions and access functions. This code is to be replicated where hash tables are used, mutatis mutandis.

```

<hashes.f90>≡
  <File header>

  module hashes

    use kinds, only: i8, i32 !NODEP!
    use bytes

    <Standard module head>

    <Hashes: public>

    contains

    <Hashes: procedures>

  end module hashes

```

### 2.8.1 The hash function

This is the one-at-a-time hash function by Bob Jenkins (from Wikipedia), re-implemented in Fortran. The function works on an array of bytes (8-bit integers), as could be produced by, e.g., the **transfer** function, and returns a single 32-bit integer. For determining the position in a hashtable, one can pick the lower bits of the result as appropriate to the hashtable size (which should be a power of 2). Note that we are working on signed integers, so the interpretation of values differs from the C version. This should not matter in practice, however.

```

<Hashes: public>≡
  public :: hash

<Hashes: procedures>≡
  function hash (key) result (hashval)
    integer(i32) :: hashval
    integer(i8), dimension(:), intent(in) :: key
    type(word32_t) :: w
    integer :: i
    w = 0_i32
    do i = 1, size (key)
      w = w + key(i)
    end do
  end function hash

```

```

        w = w + ishft (w, 10)
        w = ieor (w, ishft (w, -6))
    end do
    w = w + ishft (w, 3)
    w = ieor (w, ishft (w, -11))
    w = w + ishft (w, 15)
    hashval = w
end function hash

```

## 2.8.2 The hash table

We define a generic hashtable type (that depends on the `hash_data_t` type) together with associated methods.

This is a template:

```

<Hashtables: types>≡
    type :: hash_data_t
        integer :: i
    end type hash_data_t

```

Associated methods:

```

<Hashtables: procedures>≡
    subroutine hash_data_final (data)
        type(hash_data_t), intent(inout) :: data
    end subroutine hash_data_final

    subroutine hash_data_write (data, unit)
        type(hash_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write (u, *) data%i
    end subroutine hash_data_write

```

Each hash entry stores the unmasked hash value, the key, and points to actual data if present. Note that this could be an allocatable scalar in principle, but making it a pointer avoids deep copy when expanding the hashtable.

```

<Hashtables: types>+≡
    type :: hash_entry_t
        integer(i32) :: hashval = 0
        integer(i8), dimension(:), allocatable :: key
        type(hash_data_t), pointer :: data => null ()
    end type hash_entry_t

```

The hashtable object holds the actual table, the number of filled entries and the number of entries after which the size should be doubled. The mask is equal to the table size minus one and thus coincides with the upper bound of the table index, which starts at zero.

```

<Hashtables: types>+≡
    type :: hashtable_t
        integer :: n_entries = 0

```

```

    real :: fill_ratio = 0
    integer :: n_entries_max = 0
    integer(i32) :: mask = 0
    type(hash_entry_t), dimension(:), allocatable :: entry
end type hashtable_t

```

Initializer: The size has to be a power of two, the fill ratio is a real (machine-default!) number between 0 and 1.

```

<Hashtables: procedures> +=
subroutine hashtable_init (hashtable, size, fill_ratio)
    type(hashtable_t), intent(out) :: hashtable
    integer, intent(in) :: size
    real, intent(in) :: fill_ratio
    hashtable%fill_ratio = fill_ratio
    hashtable%n_entries_max = size * fill_ratio
    hashtable%mask = size - 1
    allocate (hashtable%entry (0:hashtable%mask))
end subroutine hashtable_init

```

Finalizer: This calls a `hash_data_final` subroutine which must exist.

```

<Hashtables: procedures> +=
subroutine hashtable_final (hashtable)
    type(hashtable_t), intent(inout) :: hashtable
    integer :: i
    do i = 0, hashtable%mask
        if (associated (hashtable%entry(i)%data)) then
            call hash_data_final (hashtable%entry(i)%data)
            deallocate (hashtable%entry(i)%data)
        end if
    end do
    deallocate (hashtable%entry)
end subroutine hashtable_final

```

Output. Here, we refer to a `hash_data_write` subroutine.

```

<Hashtables: procedures> +=
subroutine hashtable_write (hashtable, unit)
    type(hashtable_t), intent(in) :: hashtable
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    do i = 0, hashtable%mask
        if (associated (hashtable%entry(i)%data)) then
            write (u, *) i, "(hash =", hashtable%entry(i)%hashval, ")", &
                hashtable%entry(i)%key
            call hash_data_write (hashtable%entry(i)%data, unit)
        end if
    end do
end subroutine hashtable_write

```

### 2.8.3 Hashtable insertion

Insert a single entry with the hash value as trial place. If the table is filled, first expand it.

```
(Hashtables: procedures) +=
  subroutine hashtable_insert (hashtable, key, data)
    type(hashtable_t), intent(inout) :: hashtable
    integer(i8), dimension(:), intent(in) :: key
    type(hash_data_t), intent(in), target :: data
    integer(i32) :: h
    if (hashtable%n_entries >= hashtable%n_entries_max) &
      call hashtable_expand (hashtable)
    h = hash (key)
    call hashtable_insert_rec (hashtable, h, h, key, data)
  end subroutine hashtable_insert
```

We need this auxiliary routine for doubling the size of the hashtable. We rely on the fact that default assignment copies the data pointer, not the data themselves. The temporary array must not be finalized; it is deallocated automatically together with its allocatable components.

```
(Hashtables: procedures) +=
  subroutine hashtable_expand (hashtable)
    type(hashtable_t), intent(inout) :: hashtable
    type(hash_entry_t), dimension(:), allocatable :: table_tmp
    integer :: i, s
    allocate (table_tmp (0:hashtable%mask))
    table_tmp = hashtable%entry
    deallocate (hashtable%entry)
    s = 2 * size (table_tmp)
    hashtable%n_entries = 0
    hashtable%n_entries_max = s * hashtable%fill_ratio
    hashtable%mask = s - 1
    allocate (hashtable%entry (0:hashtable%mask))
    do i = 0, ubound (table_tmp, 1)
      if (associated (table_tmp(i)%data)) then
        call hashtable_insert_rec (hashtable, table_tmp(i)%hashval, &
          table_tmp(i)%hashval, table_tmp(i)%key, table_tmp(i)%data)
      end if
    end do
  end subroutine hashtable_expand
```

Insert a single entry at a trial place *h*, reduced to the table size. Collision resolution is done simply by choosing the next element, recursively until the place is empty. For bookkeeping, we preserve the original hash value. For a good hash function, there should be no clustering.

Note that if the new key exactly matches an existing key, nothing is done.

```
(Hashtables: procedures) +=
  recursive subroutine hashtable_insert_rec (hashtable, h, hashval, key, data)
    type(hashtable_t), intent(inout) :: hashtable
    integer(i32), intent(in) :: h, hashval
    integer(i8), dimension(:), intent(in) :: key
    type(hash_data_t), intent(in), target :: data
```

```

integer(i32) :: i
i = iand (h, hashtable%mask)
if (associated (hashtable%entry(i)%data)) then
    if (size (hashtable%entry(i)%key) /= size (key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
    else if (any (hashtable%entry(i)%key /= key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
    end if
else
    hashtable%entry(i)%hashval = hashval
    allocate (hashtable%entry(i)%key (size (key)))
    hashtable%entry(i)%key = key
    hashtable%entry(i)%data => data
    hashtable%n_entries = hashtable%n_entries + 1
end if
end subroutine hashtable_insert_rec

```

## 2.8.4 Hashtable lookup

The lookup function has to parallel the insert function. If the place is filled, check if the key matches. Yes: return the pointer; no: increment the hash value and check again.

```

<Hashtables: procedures>+=
function hashtable_lookup (hashtable, key) result (ptr)
    type(hash_data_t), pointer :: ptr
    type(hashtable_t), intent(in) :: hashtable
    integer(i8), dimension(:), intent(in) :: key
    ptr => hashtable_lookup_rec (hashtable, hash (key), key)
end function hashtable_lookup

<Hashtables: procedures>+=
recursive function hashtable_lookup_rec (hashtable, h, key) result (ptr)
    type(hash_data_t), pointer :: ptr
    type(hashtable_t), intent(in) :: hashtable
    integer(i32), intent(in) :: h
    integer(i8), dimension(:), intent(in) :: key
    integer(i32) :: i
    i = iand (h, hashtable%mask)
    if (associated (hashtable%entry(i)%data)) then
        if (size (hashtable%entry(i)%key) == size (key)) then
            if (all (hashtable%entry(i)%key == key)) then
                ptr => hashtable%entry(i)%data
            else
                ptr => hashtable_lookup_rec (hashtable, h + 1, key)
            end if
        else
            ptr => hashtable_lookup_rec (hashtable, h + 1, key)
        end if
    else
        ptr => null ()
    end if
end function hashtable_lookup_rec

```



```

<Hashtables: public>≡
    public :: hashtable_test

<Hashtables: procedures>+≡
    subroutine hashtable_test ()
        type(hash_data_t), pointer :: data
        type(hashtable_t) :: hashtable
        integer(i8) :: i
        call hashtable_init (hashtable, 16, 0.25)
        do i = 1, 10
            allocate (data)
            data%i = i*i
            call hashtable_insert (hashtable, (/i, i+i/), data)
        end do
        call hashtable_insert (hashtable, (/2_i8, 4_i8/), data)
        call hashtable_write (hashtable)
        data => hashtable_lookup (hashtable, (/5_i8, 10_i8/))
        if (associated (data)) then
            print *, "lookup:", data%i
        else
            print *, "lookup: --"
        end if
        data => hashtable_lookup (hashtable, (/6_i8, 12_i8/))
        if (associated (data)) then
            print *, "lookup:", data%i
        else
            print *, "lookup: --"
        end if
        data => hashtable_lookup (hashtable, (/4_i8, 9_i8/))
        if (associated (data)) then
            print *, "lookup:", data%i
        else
            print *, "lookup: --"
        end if
        call hashtable_final (hashtable)
    end subroutine hashtable_test

```

## 2.9 Message and signal handling

We are not so ambitious as to do proper exception handling in WHIZARD, but at least it may be useful to have a common interface for diagnostics: Results, messages, warnings, and such. As module variables we keep a buffer where the current message may be written to and a level indicator which tells which messages should be written on screen and which ones should be skipped. Alternatively, a string may be directly supplied to the message routine: this overrides the buffer, avoiding the necessity of formatted I/O in trivial cases.

```

<diagnostics.f90>≡
    <File header>

    module diagnostics

```

```

    use iso_c_binding !NODEP!
    use system_dependencies !NODEP!
    <Use kinds>
    <Use strings>
    use limits, only: BUFFER_SIZE, MAX_ERRORS !NODEP!
    use file_utils !NODEP!
    use iso_fortran_env, only: &
        stdout => output_unit, stdin => input_unit !NODEP!

    <Standard module head>

    <Diagnostics: public>

    <Diagnostics: parameters>

    <Diagnostics: types>

    <Diagnostics: variables>

    <Diagnostics: interfaces>

contains

    <Diagnostics: procedures>

end module diagnostics
Diagnostics levels:
<Diagnostics: parameters>≡
    integer, parameter :: &
        & TERMINATE=-2, BUG=-1, &
        & FATAL=1, ERROR=2, WARNING=3, MESSAGE=4, RESULT=5, DEBUG=6
<Diagnostics: variables>≡
    integer, save :: msg_level = RESULT
Mask fatal errors so that are treated as normal errors. Useful for interactive
mode.
<Diagnostics: public>≡
    public :: mask_fatal_errors
<Diagnostics: variables>+≡
    logical, save :: mask_fatal_errors = .false.
How to handle bugs and unmasked fatal errors. Either execute a normal stop
statement, or call the C exit() function, or try to cause a program crash by
dereferencing a null pointer.
<Diagnostics: parameters>+≡
    integer, parameter :: TERM_STOP = 0, TERM_EXIT = 1, TERM_CRASH = 2
<Diagnostics: variables>+≡
    integer, save :: handle_fatal_errors = TERM_EXIT
Keep track of errors. This might be used for exception handling, later. The
counter is incremented only for screen messages, to avoid double counting.
<Diagnostics: public>+≡
    public :: msg_count

```

```

<Diagnostics: variables>+≡
    integer, dimension(TERMINATE:DEBUG), save :: msg_count = 0

```

Keep a list of all errors and warnings. Since we do not know the number of entries beforehand, we use a linked list.

```

<Diagnostics: types>≡
    type :: string_list
        character(len=BUFFER_SIZE) :: string
        type(string_list), pointer :: next
    end type string_list
    type :: string_list_pointer
        type(string_list), pointer :: first, last
    end type string_list_pointer

<Diagnostics: variables>+≡
    type(string_list_pointer), dimension(TERMINATE:WARNING), save :: &
        & msg_list = string_list_pointer (null(), null())

```

Add the current message buffer contents to the internal list.

```

<Diagnostics: procedures>≡
    subroutine msg_add (level)
        integer, intent(in) :: level
        type(string_list), pointer :: message
        select case (level)
        case (TERMINATE:WARNING)
            allocate (message)
            message%string = msg_buffer
            nullify (message%next)
            if (.not.associated (msg_list(level)%first)) &
                & msg_list(level)%first => message
            if (associated (msg_list(level)%last)) &
                & msg_list(level)%last%next => message
            msg_list(level)%last => message
            msg_count(level) = msg_count(level) + 1
        end select
    end subroutine msg_add

```

Initialization:

```

<Diagnostics: public>+≡
    public :: msg_list_clear

<Diagnostics: procedures>+≡
    subroutine msg_list_clear
        integer :: level
        type(string_list), pointer :: message
        do level = TERMINATE, WARNING
            do while (associated (msg_list(level)%first))
                message => msg_list(level)%first
                msg_list(level)%first => message%next
                deallocate (message)
            end do
            nullify (msg_list(level)%last)
        end do
        msg_count = 0
    end subroutine msg_list_clear

```

Display the summary of errors and warnings (no need to count fatals...)

```

<Diagnostics: public>+≡
    public :: msg_summary

<Diagnostics: procedures>+≡
    subroutine msg_summary (unit)
        integer, intent(in), optional :: unit
        call expect_summary (unit)
1    format (A,1x,I2,1x,A,I2,1x,A)
        if (msg_count(ERROR) > 0 .and. msg_count(WARNING) > 0) then
            write (msg_buffer, 1) "There were", &
                & msg_count(ERROR), "error(s) and ", &
                & msg_count(WARNING), "warning(s)."
            call msg_message (unit=unit)
        else if (msg_count(ERROR) > 0) then
            write (msg_buffer, 1) "There were", &
                & msg_count(ERROR), "error(s) and no warnings."
            call msg_message (unit=unit)
        else if (msg_count(WARNING) > 0) then
            write (msg_buffer, 1) "There were no errors and ", &
                & msg_count(WARNING), "warning(s)."
            call msg_message (unit=unit)
        end if
    end subroutine msg_summary

```

Print the list of all messages of a given level.

```

<Diagnostics: public>+≡
    public :: msg_listing

<Diagnostics: procedures>+≡
    subroutine msg_listing (level, unit, prefix)
        integer, intent(in) :: level
        integer, intent(in), optional :: unit
        character(len=*), intent(in), optional :: prefix
        type(string_list), pointer :: message
        integer :: u
        u = output_unit (unit); if (u < 0) return
        if (present (unit)) u = unit
        message => msg_list(level)%first
        do while (associated (message))
            if (present (prefix)) then
                write (u, "(A)") prefix // trim (message%string)
            else
                write (u, "(A)") trim (message%string)
            end if
            message => message%next
        end do
        flush (u)
    end subroutine msg_listing

```

There is a hard limit on the line length which we should export. This buffer size is used both by the message handler and by the lexer below.

```

<Limits: public parameters>+≡
    integer, parameter, public :: BUFFER_SIZE = 1000

```

The message buffer:

```

<Diagnostics: public>+≡
    public :: msg_buffer

<Diagnostics: variables>+≡
    character(len=BUFFER_SIZE), save :: msg_buffer = " "

```

After a message is issued, the buffer should be cleared:

```

<Diagnostics: procedures>+≡
    subroutine buffer_clear
        msg_buffer = " "
    end subroutine buffer_clear

```

The generic handler for messages. If the unit is omitted (or = 6), the message is written to standard output if the precedence is sufficiently high (as determined by the value of `msg_level`). If the string is omitted, the buffer is used. In any case, the buffer is cleared after printing. In accordance with FORTRAN custom, the first column in the output is left blank. For messages and warnings, an additional exclamation mark and a blank is prepended. Furthermore, each message is appended to the internal message list (without prepending anything).

```

<Diagnostics: procedures>+≡
    subroutine message_print (level, string, str_arr, unit, logfile)
        integer, intent(in) :: level
        character(len=*), intent(in), optional :: string
        type(string_t), dimension(:), intent(in), optional :: str_arr
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: logfile
        type(string_t) :: prep_string, aux_string, head_footer
        integer :: lu, i
        logical :: severe, is_error
        severe = .false.
        head_footer = "*****"
        aux_string = ""
        is_error = .false.
    !   integer :: proc_id
    !   call mpi90_rank (proc_id)
    !   if (proc_id==WHIZARD_ROOT) then
        select case (level)
        case (TERMINATE)
            prep_string = ""
        case (BUG)
            prep_string = "*** WHIZARD BUG: "
            aux_string = "*****"
            severe = .true.
            is_error = .true.
        case (FATAL)
            prep_string = "*** FATAL ERROR: "
            aux_string = "*****"
            severe = .true.
            is_error = .true.
        case (ERROR)
            prep_string = "*** ERROR: "
            aux_string = "*****"
            is_error = .true.

```

```

case (WARNING)
    prep_string = "Warning: "
case (MESSAGE, DEBUG)
    prep_string = "| "
case default
    prep_string = ""
end select
if (present(string)) msg_buffer = string
lu = log_unit
if (present(unit)) then
    if (unit /= stdout) then
        if (severe) write (unit, "(A)") char(head_footer)
        if (is_error) write (unit, "(A)") char(head_footer)
        write (unit, "(A,A)") char(prepare_string), trim(msg_buffer)
        if (present (str_arr)) then
            do i = 1, size(str_arr)
                write (unit, "(A,A)") char(aux_string), char(trim(str_arr(i)))
            end do
        end if
        if (is_error) write (unit, "(A)") char(head_footer)
        if (severe) write (unit, "(A)") char(head_footer)
        flush (unit)
        lu = -1
    else if (level <= msg_level) then
        if (severe) print "(A)", char(head_footer)
        if (is_error) print "(A)", char(head_footer)
        print "(A,A)", char(prepare_string), trim(msg_buffer)
        if (present (str_arr)) then
            do i = 1, size(str_arr)
                print "(A,A)", char(aux_string), char(trim(str_arr(i)))
            end do
        end if
        if (is_error) print "(A)", char(head_footer)
        if (severe) print "(A)", char(head_footer)
        flush (stdout)
        if (unit == log_unit) lu = -1
    end if
else if (level <= msg_level) then
    if (severe) print "(A)", char(head_footer)
    if (is_error) print "(A)", char(head_footer)
    print "(A,A)", char(prepare_string), trim(msg_buffer)
    if (present (str_arr)) then
        do i = 1, size(str_arr)
            print "(A,A)", char(aux_string), char(trim(str_arr(i)))
        end do
    end if
    if (is_error) print "(A)", char(head_footer)
    if (severe) print "(A)", char(head_footer)
    flush (stdout)
end if
if (present (logfile)) then
    if (.not. logfile) lu = -1
end if
if (logging .and. lu >= 0) then

```

```

        if (severe) write (lu, "(A)") char(head_footer)
        if (is_error) write (lu, "(A)") char(head_footer)
        write (lu, "(A,A)") char(prepare_string), trim(msg_buffer)
        if (present (str_arr)) then
            do i = 1, size(str_arr)
                write (lu, "(A,A)") char(aux_string), char(trim(str_arr(i)))
            end do
        end if
        if (is_error) write (lu, "(A)") char(head_footer)
        if (severe) write (lu, "(A)") char(head_footer)
        flush (lu)
    end if
!    end if
    call msg_add (level)
    call buffer_clear
end subroutine message_print

```

The specific handlers. In the case of fatal errors, bugs (failed assertions) and normal termination execution is stopped. For non-fatal errors a message is printed to standard output if no unit is given. Only if the number of `MAX_ERRORS` errors is reached, we abort the program. There are no further actions in the other cases, but this may change.

*<Diagnostics: public>+≡*

```

public :: msg_terminate
public :: msg_bug, msg_fatal, msg_error, msg_warning
public :: msg_message, msg_result, msg_debug

```

*<Diagnostics: procedures>+≡*

```

subroutine msg_terminate (string, unit, quit_code)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    integer, intent(in), optional :: quit_code
    integer(c_int) :: return_code
    call release_term_signals ()
    if (present (quit_code)) then
        return_code = quit_code
    else
        return_code = 0
    end if
    if (present (string)) &
        call message_print (MESSAGE, string, unit=unit)
    call msg_summary (unit)
    if (return_code == 0 .and. expect_failures /= 0) then
        return_code = 5
        call message_print (MESSAGE, &
            "WHIZARD run finished with 'expect' failure(s).", unit=unit)
    else if (return_code == 7) then
        call message_print (MESSAGE, &
            "WHIZARD run finished with failed self-test.", unit=unit)
    else
        call message_print (MESSAGE, "WHIZARD run finished.", unit=unit)
    end if
    call message_print (0, &
        "|=====|", unit=u

```

```

        call logfile_final ()
        if (return_code /= 0) then
            call exit (return_code)
        else
!           stop
            call exit (0)
        end if
    end subroutine msg_terminate

subroutine msg_bug (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, pointer :: crash_ptr
    call message_print (BUG, string, arr, unit)
    call msg_summary (unit)
    select case (handle_fatal_errors)
    case (TERM_EXIT)
        call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
        call exit (-1_c_int)
    case (TERM_CRASH)
        print *, "*** Intentional crash ***"
        crash_ptr => null ()
        print *, crash_ptr
    end select
    stop "WHIZARD run aborted."
end subroutine msg_bug

recursive subroutine msg_fatal (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, pointer :: crash_ptr
    if (mask_fatal_errors) then
        call msg_error (string, arr, unit)
    else
        call message_print (FATAL, string, arr, unit)
        call msg_summary (unit)
        select case (handle_fatal_errors)
        case (TERM_EXIT)
            call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
            call exit (1_c_int)
        case (TERM_CRASH)
            print *, "*** Intentional crash ***"
            crash_ptr => null ()
            print *, crash_ptr
        end select
        stop "WHIZARD run aborted."
    end if
end subroutine msg_fatal

subroutine msg_error (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string

```



```

        type(string_t), dimension(:), intent(in), optional :: arr
        call message_print (ERROR, string, arr, unit)
        if (msg_count(ERROR) >= MAX_ERRORS) then
            mask_fatal_errors = .false.
            call msg_fatal (" Too many errors encountered.")
!       else if (.not.present(unit) .and. .not.mask_fatal_errors) then
!           call message_print (MESSAGE, "                (WHIZARD run continues)")
        end if
    end subroutine msg_error

subroutine msg_warning (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    call message_print (WARNING, string, arr, unit)
end subroutine msg_warning

subroutine msg_message (string, unit, arr, logfile)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, intent(in), optional :: logfile
    call message_print (MESSAGE, string, arr, unit, logfile)
end subroutine msg_message

subroutine msg_result (string, arr, unit, logfile)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, intent(in), optional :: logfile
    call message_print (RESULT, string, arr, unit, logfile)
end subroutine msg_result

subroutine msg_debug (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    call message_print (DEBUG, string, arr, unit)
end subroutine msg_debug

```

Interface to the standard clib exit function

*<Diagnostics: interfaces>+≡*

```

interface
    subroutine exit (status) bind (C)
        use iso_c_binding !NODEP!
        integer(c_int), value :: status
    end subroutine exit
end interface

```

*<Limits: public parameters>+≡*

```

integer, parameter, public :: MAX_ERRORS = 10

```

Print the WHIZARD banner:

*<Diagnostics: public>+≡*



```

call message_print (0, "| M. Moretti, T. Ohl, J. Reuter, arXiv: hep-ph/0102195
call message_print (0, "|
call message_print (0, "|=====
call message_print (0, "| WHIZARD " // WHIZARD_VERSION, unit=uni
call message_print (0, "|=====
end subroutine msg_banner

```

### 2.9.1 Logfile

All screen output should be duplicated in the logfile, unless requested otherwise.

```

<Diagnostics: public>+≡
public :: logging

<Diagnostics: variables>+≡
integer, save :: log_unit = -1
logical, target, save :: logging = .false.

<Diagnostics: public>+≡
public :: logfile_init

<Diagnostics: procedures>+≡
subroutine logfile_init (filename)
type(string_t), intent(in) :: filename
call msg_message ("Writing log to '" // char (filename) // "'")
if (.not. logging) call msg_message ("(Logging turned off.)")
log_unit = free_unit ()
open (file = char (filename), unit = log_unit, &
      action = "write", status = "replace")
end subroutine logfile_init

<Diagnostics: public>+≡
public :: logfile_final

<Diagnostics: procedures>+≡
subroutine logfile_final ()
if (log_unit >= 0) then
close (log_unit)
log_unit = -1
end if
end subroutine logfile_final

```

This returns the valid logfile unit only if the default is write to screen, and if logfile is not set false.

```

<Diagnostics: public>+≡
public :: logfile_unit

<Diagnostics: procedures>+≡
function logfile_unit (unit, logfile)
integer :: logfile_unit
integer, intent(in), optional :: unit
logical, intent(in), optional :: logfile
if (logging) then
if (present (unit)) then
if (unit == stdout) then

```

```

        logfile_unit = log_unit
    else
        logfile_unit = -1
    end if
else if (present (logfile)) then
    if (logfile) then
        logfile_unit = log_unit
    else
        logfile_unit = -1
    end if
else
    logfile_unit = log_unit
end if
else
    logfile_unit = -1
end if
end function logfile_unit

```

## 2.9.2 Checking values

The `expect` function does not just check a value for correctness (actually, it checks if a logical expression is true); it records its result here. If failures are present when the program terminates, the exit code is nonzero.

```

<Diagnostics: variables>+≡
    integer, save :: expect_total = 0
    integer, save :: expect_failures = 0

<Diagnostics: public>+≡
    public :: expect_record

<Diagnostics: procedures>+≡
    subroutine expect_record (success)
        logical, intent(in) :: success
        expect_total = expect_total + 1
        if (.not. success) expect_failures = expect_failures + 1
    end subroutine expect_record

<Diagnostics: public>+≡
    public :: expect_clear

<Diagnostics: procedures>+≡
    subroutine expect_clear ()
        expect_total = 0
        expect_failures = 0
    end subroutine expect_clear

<Diagnostics: public>+≡
    public :: expect_summary

<Diagnostics: procedures>+≡
    subroutine expect_summary (unit)
        integer, intent(in), optional :: unit

```

```

    if (expect_total /= 0) then
        call msg_message ("Summary of value checks:", unit)
        write (msg_buffer, "(2x,A,1x,I0,1x,A,1x,A,1x,I0)") &
            "Failures:", expect_failures, "/", "Total:", expect_total
        call msg_message (unit=unit)
    end if
end subroutine expect_summary

```

Helpers for converting integers into strings with minimal length.

```

<Diagnostics: public>+≡
    public :: int2string
    public :: int2char
    public :: int2fixed

<Diagnostics: procedures>+≡
    pure function int2fixed (i) result (c)
        integer, intent(in) :: i
        character(200) :: c
        c = ""
        write (c, *) i
        c = adjustl (c)
    end function int2fixed

    pure function int2string (i) result (s)
        integer, intent(in) :: i
        type (string_t) :: s
        s = trim (int2fixed (i))
    end function int2string

    pure function int2char (i) result (c)
        integer, intent(in) :: i
        character(len (trim (int2fixed (i)))) :: c
        c = int2fixed (i)
    end function int2char

```

Dito for reals.

```

<Diagnostics: public>+≡
    public :: real2string
    public :: real2char
    public :: real2fixed

<Diagnostics: interfaces>+≡
    interface real2string
        module procedure real2string_list, real2string_fmt
    end interface
    interface real2char
        module procedure real2char_list, real2char_fmt
    end interface

<Diagnostics: procedures>+≡
    pure function real2fixed (x, fmt) result (c)
        real(default), intent(in) :: x
        character(*), intent(in), optional :: fmt
        character(200) :: c

```

```

    c = ""
    write (c, *) x
    c = adjustl (c)
end function real2fixed

pure function real2fixed_fmt (x, fmt) result (c)
    real(default), intent(in) :: x
    character(*), intent(in) :: fmt
    character(200) :: c
    c = ""
    write (c, fmt) x
    c = adjustl (c)
end function real2fixed_fmt

pure function real2string_list (x) result (s)
    real(default), intent(in) :: x
    type(string_t) :: s
    s = trim (real2fixed (x))
end function real2string_list

pure function real2string_fmt (x, fmt) result (s)
    real(default), intent(in) :: x
    character(*), intent(in) :: fmt
    type(string_t) :: s
    s = trim (real2fixed_fmt (x, fmt))
end function real2string_fmt

pure function real2char_list (x) result (c)
    real(default), intent(in) :: x
    character(len_trim (real2fixed (x))) :: c
    c = real2fixed (x)
end function real2char_list

pure function real2char_fmt (x, fmt) result (c)
    real(default), intent(in) :: x
    character(*), intent(in) :: fmt
    character(len_trim (real2fixed_fmt (x, fmt))) :: c
    c = real2fixed_fmt (x, fmt)
end function real2char_fmt

```

Dito for complex values; we do not use the slightly ugly FORTRAN output form here but instead introduce our own.

```

<Diagnostics: public>+≡
    public :: cmplx2string
    ! Ifort and Portland seem to have problems with this -> temporarily disable it
    !   public :: cmplx2char

<Diagnostics: procedures>+≡
    pure function cmplx2string (x) result (s)
        complex(default), intent(in) :: x
        type(string_t) :: s
        s = real2string (real (x, default))
        if (aimag (x) /= 0) s = s // " + " // real2string (aimag (x)) // " I"
    end function cmplx2string

```

```

pure function cmplx2char (x) result (c)
  complex(default), intent(in) :: x
  character(len (char (cmplx2string (x)))) :: c
  c = char (cmplx2string (x))
end function cmplx2char

```

### 2.9.3 Signal handling

Killing the program by external signals may leave the files written by it in an undefined state. This can be avoided by catching signals and deferring program termination. Instead of masking only critical sections, we choose to mask signals globally (done in the main program) and terminate the program at predefined checkpoints only. Checkpoints are after each command, within the sampling function (so the program can be terminated after each event), and after each iteration in the phase-space generation algorithm.

Signal handling is done via a C interface to the `sigaction` system call. When a signal is raised that has been masked by the handler, the corresponding variable is set to the value of the signal. The variables are visible from the C signal handler.

The signal `SIGINT` is for keyboard interrupt (ctrl-C), `SIGTERM` is for system interrupt, e.g., at shutdown. The `SIGXCPU` and `SIGXFSZ` signals may be issued by batch systems.

```

<Diagnostics: public>+≡
  public :: wo_sigint
  public :: wo_sigterm
  public :: wo_sigxcpu
  public :: wo_sigxfsz

<Diagnostics: variables>+≡
  integer(c_int), bind(C), volatile :: wo_sigint = 0
  integer(c_int), bind(C), volatile :: wo_sigterm = 0
  integer(c_int), bind(C), volatile :: wo_sigxcpu = 0
  integer(c_int), bind(C), volatile :: wo_sigxfsz = 0

```

Here are the interfaces to the C functions. The routine `mask_term_signals` forces termination signals to be delayed. `release_term_signals` restores normal behavior. However, the program can be terminated anytime by calling `terminate_now_if_signal` which inspects the signals and terminates the program if requested..

```

<Diagnostics: public>+≡
  public :: mask_term_signals

<Diagnostics: procedures>+≡
  subroutine mask_term_signals ()
    integer(c_int) :: status
    logical :: ok
    wo_sigint = 0
    ok = wo_mask_sigint () == 0
    if (.not. ok) call msg_error ("Masking SIGINT failed")
    wo_sigterm = 0

```

```

ok = wo_mask_sigterm () == 0
if (.not. ok) call msg_error ("Masking SIGTERM failed")
wo_sigxcpu = 0
ok = wo_mask_sigxcpu () == 0
if (.not. ok) call msg_error ("Masking SIGXCPU failed")
wo_sigxfsz = 0
ok = wo_mask_sigxfsz () == 0
if (.not. ok) call msg_error ("Masking SIGXFSZ failed")
end subroutine mask_term_signals

```

*<Diagnostics: interfaces>+≡*

```

interface
  integer(c_int) function wo_mask_sigint () bind(C)
  import
end function wo_mask_sigint
end interface
interface
  integer(c_int) function wo_mask_sigterm () bind(C)
  import
end function wo_mask_sigterm
end interface
interface
  integer(c_int) function wo_mask_sigxcpu () bind(C)
  import
end function wo_mask_sigxcpu
end interface
interface
  integer(c_int) function wo_mask_sigxfsz () bind(C)
  import
end function wo_mask_sigxfsz
end interface

```

*<Diagnostics: public>+≡*

```

public :: release_term_signals

```

*<Diagnostics: procedures>+≡*

```

subroutine release_term_signals ()
  integer(c_int) :: status
  logical :: ok
  ok = wo_release_sigint () == 0
  if (.not. ok) call msg_error ("Releasing SIGINT failed")
  ok = wo_release_sigterm () == 0
  if (.not. ok) call msg_error ("Releasing SIGTERM failed")
  ok = wo_release_sigxcpu () == 0
  if (.not. ok) call msg_error ("Releasing SIGXCPU failed")
  ok = wo_release_sigxfsz () == 0
  if (.not. ok) call msg_error ("Releasing SIGXFSZ failed")
end subroutine release_term_signals

```

*<Diagnostics: interfaces>+≡*

```

interface
  integer(c_int) function wo_release_sigint () bind(C)
  import
end function wo_release_sigint

```



```

end interface
interface
    integer(c_int) function wo_release_sigterm () bind(C)
    import
    end function wo_release_sigterm
end interface
interface
    integer(c_int) function wo_release_sigxcpu () bind(C)
    import
    end function wo_release_sigxcpu
end interface
interface
    integer(c_int) function wo_release_sigxfsz () bind(C)
    import
    end function wo_release_sigxfsz
end interface

<Diagnostics: public>+≡
    public :: terminate_now_if_signal

<Diagnostics: procedures>+≡
    subroutine terminate_now_if_signal ()
        if (wo_sigint /= 0) then
            call msg_terminate ("Signal SIGINT (keyboard interrupt) received.", &
                quit_code=int (wo_sigint))
        else if (wo_sigterm /= 0) then
            call msg_terminate ("Signal SIGTERM (termination signal) received.", &
                quit_code=int (wo_sigterm))
        else if (wo_sigxcpu /= 0) then
            call msg_terminate ("Signal SIGXCPU (CPU time limit exceeded) received.", &
                quit_code=int (wo_sigxcpu))
        else if (wo_sigxfsz /= 0) then
            call msg_terminate ("Signal SIGXFSZ (file size limit exceeded) received.", &
                quit_code=int (wo_sigxfsz))
        end if
    end subroutine terminate_now_if_signal

```

## 2.10 C wrapper for sigaction

This implements calls to `sigaction` and the appropriate signal handlers in C.

```

<signal_interface.c>≡
/*
<File header>
*/
#include <signal.h>
#include <stdlib.h>

extern int wo_sigint;
extern int wo_sigterm;
extern int wo_sigxcpu;
extern int wo_sigxfsz;

```

```

static void wo_handler_sigint (int sig) {
    wo_sigint = sig;
}

static void wo_handler_sigterm (int sig) {
    wo_sigterm = sig;
}

static void wo_handler_sigxcpu (int sig) {
    wo_sigxcpu = sig;
}

static void wo_handler_sigxfsz (int sig) {
    wo_sigxfsz = sig;
}

int wo_mask_sigint () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = wo_handler_sigint;
    return sigaction(SIGINT, &sa, NULL);
}

int wo_mask_sigterm () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = wo_handler_sigterm;
    return sigaction(SIGTERM, &sa, NULL);
}

int wo_mask_sigxcpu () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = wo_handler_sigxcpu;
    return sigaction(SIGXCPU, &sa, NULL);
}

int wo_mask_sigxfsz () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = wo_handler_sigxfsz;
    return sigaction(SIGXFSZ, &sa, NULL);
}

```

```

}

int wo_release_sigint () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = SIG_DFL;
    return sigaction(SIGINT, &sa, NULL);
}

int wo_release_sigterm () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = SIG_DFL;
    return sigaction(SIGTERM, &sa, NULL);
}

int wo_release_sigxcpu () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = SIG_DFL;
    return sigaction(SIGXCPU, &sa, NULL);
}

int wo_release_sigxfsz () {
    struct sigaction sa;
    sigset_t blocks;
    sigfillset (&blocks);
    sa.sa_flags = 0;
    sa.sa_mask = blocks;
    sa.sa_handler = SIG_DFL;
    return sigaction(SIGXFSZ, &sa, NULL);
}

```

## 2.11 C wrapper for printf

The `printf` family of functions is implemented in C with an undefined number of arguments. This is not supported by the `bind(C)` interface. We therefore write wrappers for the versions of `sprintf` that we will actually use.

```

<sprintf_interface.c>≡
/*
  <File header>
*/
#include <stdio.h>

```

```

int sprintf_none(char* str, const char* format) {
    return sprintf(str, format);
}

int sprintf_int(char* str, const char* format, int val) {
    return sprintf(str, format, val);
}

int sprintf_double(char* str, const char* format, double val) {
    return sprintf(str, format, val);
}

int sprintf_str(char* str, const char* format, const char* val) {
    return sprintf(str, format, val);
}

```

*(sprintf interfaces)*≡

```

interface
    function sprintf_none (str, fmt) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
    end function sprintf_none
end interface

interface
    function sprintf_int (str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        integer(c_int), value :: val
    end function sprintf_int
end interface

interface
    function sprintf_double (str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        real(c_double), value :: val
    end function sprintf_double
end interface

interface
    function sprintf_str(str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        character(c_char), dimension(*), intent(in) :: val
    end function sprintf_str
end interface

```

```

    end function sprintf_str
end interface

```

## 2.12 Interface for formatted I/O

For access to formatted printing (possibly input), we interface the C `printf` family of functions. There are two important issues here:

1. `printf` takes an arbitrary number of arguments, relying on the C stack. This is not interoperable. We interface it with C wrappers that output a single integer, real or string and restrict the allowed formats accordingly.
2. Restricting format strings is essential also for preventing format string attacks. Allowing arbitrary format string would create a real security hole in a Fortran program.
3. The string returned by `sprintf` must be allocated to the right size.

```

<formats.f90>≡
  <File header>

```

```

module formats

```

```

    use iso_c_binding !NODEP!
    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!

```

```

  <Standard module head>

```

```

  <Formats: public>

```

```

  <Formats: parameters>

```

```

  <Formats: types>

```

```

  <Formats: interfaces>

```

```

contains

```

```

  <Formats: procedures>

```

```

end module formats

```

### 2.12.1 Parsing a C format string

The C format string contains characters and format conversion specifications. The latter are initiated by a % sign. If the next letter is also a %, a percent sign is printed and no conversion is done. Otherwise, a conversion is done and applied to the next argument in the argument list. First comes an optional flag (#, 0, -, ,

+, or space), an optional field width (decimal digits starting not with zero), an optional precision (period, then another decimal digit string), a length modifier (irrelevant for us, therefore not supported), and a conversion specifier: **d** or **i** for integer; **e**, **f**, **g** (also upper case) for double-precision real, **s** for a string.

We explicitly exclude all other conversion specifiers, and we check the specifiers against the actual arguments.

## A type for passing arguments

This is a polymorphic type that can hold integer, real (double), and string arguments.

```
(Formats: parameters)≡
  integer, parameter :: ARGTYPE_NONE = 0
  integer, parameter :: ARGTYPE_LOG = 1
  integer, parameter :: ARGTYPE_INT = 2
  integer, parameter :: ARGTYPE_REAL = 3
  integer, parameter :: ARGTYPE_STR = 4
```

The integer and real entries are actually scalars, but we avoid relying on the allocatable-scalar feature and make them one-entry arrays. The character entry is a real array which is a copy of the string.

Logical values are mapped to strings (true or false), so this type parameter value is mostly unused.

```
(Formats: public)≡
  public :: sprintf_arg_t
```

```
(Formats: types)≡
  type :: sprintf_arg_t
  private
    integer :: type = ARGTYPE_NONE
    integer(c_int), dimension(:), allocatable :: ival
    real(c_double), dimension(:), allocatable :: rval
    character(c_char), dimension(:), allocatable :: sval
  end type sprintf_arg_t
```

```
(Formats: public)+≡
  public :: sprintf_arg_init
```

```
(Formats: interfaces)≡
  interface sprintf_arg_init
    module procedure sprintf_arg_init_log
    module procedure sprintf_arg_init_int
    module procedure sprintf_arg_init_real
    module procedure sprintf_arg_init_str
  end interface
```

```
(Formats: procedures)≡
  subroutine sprintf_arg_init_log (arg, lval)
    type(sprintf_arg_t), intent(out) :: arg
    logical, intent(in) :: lval
    arg%type = ARGTYPE_STR
    if (lval) then
      allocate (arg%sval (5))
```

```

        arg%sval = (/ 't', 'r', 'u', 'e', c_null_char /)
    else
        allocate (arg%sval (6))
        arg%sval = (/ 'f', 'a', 'l', 's', 'e', c_null_char /)
    end if
end subroutine sprintf_arg_init_log

subroutine sprintf_arg_init_int (arg, ival)
    type(sprintf_arg_t), intent(out) :: arg
    integer, intent(in) :: ival
    arg%type = ARGTYPE_INT
    allocate (arg%ival (1))
    arg%ival = ival
end subroutine sprintf_arg_init_int

subroutine sprintf_arg_init_real (arg, rval)
    type(sprintf_arg_t), intent(out) :: arg
    real(default), intent(in) :: rval
    arg%type = ARGTYPE_REAL
    allocate (arg%rval (1))
    arg%rval = rval
end subroutine sprintf_arg_init_real

subroutine sprintf_arg_init_str (arg, sval)
    type(sprintf_arg_t), intent(out) :: arg
    type(string_t), intent(in) :: sval
    integer :: i
    arg%type = ARGTYPE_STR
    allocate (arg%sval (len (sval) + 1))
    do i = 1, len (sval)
        arg%sval(i) = extract (sval, i, i)
    end do
    arg%sval(len (sval) + 1) = c_null_char
end subroutine sprintf_arg_init_str

<Formats: procedures>+≡
subroutine sprintf_arg_write (arg, unit)
    type(sprintf_arg_t), intent(in) :: arg
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    select case (arg%type)
    case (ARGTYPE_NONE)
        write (u, *) "[none]"
    case (ARGTYPE_INT)
        write (u, "(1x,A,1x)", advance = "no") "[int]"
        write (u, *) arg%ival
    case (ARGTYPE_REAL)
        write (u, "(1x,A,1x)", advance = "no") "[real]"
        write (u, *) arg%rval
    case (ARGTYPE_STR)
        write (u, "(1x,A,1x,A)", advance = "no") "[string]", ""
        write (u, *) arg%rval, ""
    end select
end subroutine

```

```
end subroutine sprintf_arg_write
```

Return an upper bound for the length of the printed version; in case of strings the result is exact.

*(Formats: procedures)+≡*

```
elemental function sprintf_arg_get_length (arg) result (length)
  integer :: length
  type(sprintf_arg_t), intent(in) :: arg
  select case (arg%type)
  case (ARGTYPE_INT)
    length = log10 (real (huge (arg%ival(1)))) + 2
  case (ARGTYPE_REAL)
    length = log10 (real (radix (arg%rval(1))) ** digits (arg%rval(1))) + 8
  case (ARGTYPE_STR)
    length = size (arg%sval)
  case default
    length = 0
  end select
end function sprintf_arg_get_length
```

*(Formats: procedures)+≡*

```
subroutine sprintf_arg_apply_printf (arg, fmt, result, actual_length)
  type(sprintf_arg_t), intent(in) :: arg
  character(c_char), dimension(:), intent(in) :: fmt
  character(c_char), dimension(:), intent(inout) :: result
  integer, intent(out) :: actual_length
  integer(c_int) :: ival
  real(c_double) :: rval
  select case (arg%type)
  case (ARGTYPE_NONE)
    actual_length = sprintf_none (result, fmt)
  case (ARGTYPE_INT)
    ival = arg%ival(1)
    actual_length = sprintf_int (result, fmt, ival)
  case (ARGTYPE_REAL)
    rval = arg%rval(1)
    actual_length = sprintf_double (result, fmt, rval)
  case (ARGTYPE_STR)
    actual_length = sprintf_str (result, fmt, arg%sval)
  case default
    call msg_bug ("sprintf_arg_apply_printf called with illegal type")
  end select
  if (actual_length < 0) then
    write (msg_buffer, *) "Format: '", fmt, "'"
    call msg_message ()
    write (msg_buffer, *) "Output: '", result, "'"
    call msg_message ()
    call msg_error ("I/O error in sprintf call")
    actual_length = 0
  end if
end subroutine sprintf_arg_apply_printf
```



## Container type for the output

There is a procedure which chops the format string into pieces that contain at most one conversion specifier. Pairing this with a `sprintf_arg` object, we get the actual input to the `sprintf` interface. The type below holds this input and can allocate the output string.

*(Formats: types)+≡*

```
type :: sprintf_interface_t
  private
  character(c_char), dimension(:), allocatable :: input_fmt
  type(sprintf_arg_t) :: arg
  character(c_char), dimension(:), allocatable :: output_str
  integer :: output_str_len = 0
end type sprintf_interface_t
```

*(Formats: procedures)+≡*

```
subroutine sprintf_interface_init (intf, fmt, arg)
  type(sprintf_interface_t), intent(out) :: intf
  type(string_t), intent(in) :: fmt
  type(sprintf_arg_t), intent(in) :: arg
  integer :: fmt_len, i
  fmt_len = len (fmt)
  allocate (intf%input_fmt (fmt_len + 1))
  do i = 1, fmt_len
    intf%input_fmt(i) = extract (fmt, i, i)
  end do
  intf%input_fmt(fmt_len+1) = c_null_char
  intf%arg = arg
  allocate (intf%output_str (len (fmt) + sprintf_arg_get_length (arg) + 1))
end subroutine sprintf_interface_init
```

*(Formats: procedures)+≡*

```
subroutine sprintf_interface_write (intf, unit)
  type(sprintf_interface_t), intent(in) :: intf
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, *) "Format string = ", "'", intf%input_fmt, "'"
  write (u, "(1x,A,1x)", advance = "no") "Argument = "
  call sprintf_arg_write (intf%arg, unit)
  if (intf%output_str_len > 0) then
    write (u, *) "Result string = ", &
      "'", intf%output_str (1:intf%output_str_len), "'"
  end if
end subroutine sprintf_interface_write
```

Return the output string:

*(Formats: procedures)+≡*

```
function sprintf_interface_get_result (intf) result (string)
  type(string_t) :: string
  type(sprintf_interface_t), intent(in) :: intf
  character(kind = c_char, len = max (intf%output_str_len, 0)) :: buffer
```

```

integer :: i
if (intf%output_str_len > 0) then
  do i = 1, intf%output_str_len
    buffer(i:i) = intf%output_str(i)
  end do
  string = buffer(1:intf%output_str_len)
else
  string = ""
end if
end function sprintf_interface_get_result

```

*<Formats: procedures>+≡*

```

subroutine sprintf_interface_apply_sprintf (intf)
  type(sprintf_interface_t), intent(inout) :: intf
  call sprintf_arg_apply_sprintf &
    (intf%arg, intf%input_fmt, intf%output_str, intf%output_str_len)
end subroutine sprintf_interface_apply_sprintf

```

Import the interfaces defined in the previous section:

*<Formats: interfaces>+≡*  
*<sprintf interfaces>*

## Scan the format string

Chop it into pieces that contain one conversion specifier each. The zero-th piece contains the part before the first specifier. Check the specifiers and allow only the subset that we support. Also check for an exact match between conversion specifiers and input arguments. The result is an allocated array of `sprintf_interface` object; each one contains a piece of the format string and the corresponding argument.

*<Formats: procedures>+≡*

```

subroutine chop_and_check_format_string (fmt, arg, intf)
  type(string_t), intent(in) :: fmt
  type(sprintf_arg_t), dimension(:), intent(in) :: arg
  type(sprintf_interface_t), dimension(:), intent(out), allocatable :: intf
  integer :: n_args, i
  type(string_t), dimension(:), allocatable :: split_fmt
  type(string_t) :: word, buffer, separator
  integer :: pos, length, l
  logical :: ok
  type(sprintf_arg_t) :: arg_null
  ok = .true.
  length = 0
  n_args = size (arg)
  allocate (split_fmt (0:n_args))
  split_fmt = ""
  buffer = fmt
  SCAN_ARGS: do i = 1, n_args
    FIND_CONVERSION: do
      call split (buffer, word, "%", separator=separator)
      if (separator == "") then

```

```

        call msg_message (''' // char (fmt) // ''')
        call msg_error ("C-formatting string: " &
            // "too few conversion specifiers in format string")
        ok = .false.; exit SCAN_ARGS
    end if
    split_fmt(i-1) = split_fmt(i-1) // word
    if (extract (buffer, 1, 1) /= "%") then
        split_fmt(i) = "%"
        exit FIND_CONVERSION
    else
        split_fmt(i-1) = split_fmt(i-1) // "%"
    end if
end do FIND_CONVERSION
pos = verify (buffer, "#0-- ") ! Flag characters (zero or more)
split_fmt(i) = split_fmt(i) // extract (buffer, 1, pos-1)
buffer = remove (buffer, 1, pos-1)
pos = verify (buffer, "123456890") ! Field width
word = extract (buffer, 1, pos-1)
if (len (word) /= 0) then
    call read_int_from_string (word, len (word), 1)
    length = length + 1
end if
split_fmt(i) = split_fmt(i) // word
buffer = remove (buffer, 1, pos-1)
if (extract (buffer, 1, 1) == ".") then
    buffer = remove (buffer, 1, 1)
    pos = verify (buffer, "1234567890") ! Precision
    split_fmt(i) = split_fmt(i) // "." // extract (buffer, 1, pos-1)
    buffer = remove (buffer, 1, pos-1)
end if
! Length modifier would come here, but is not allowed
select case (char (extract (buffer, 1, 1))) ! conversion specifier
case ("d", "i")
    if (arg(i)%type /= ARGTYPE_INT) then
        call msg_message (''' // char (fmt) // ''')
        call msg_error ("C-formatting string: " &
            // "argument type mismatch: integer value expected")
        ok = .false.; exit SCAN_ARGS
    end if
case ("e", "E", "f", "F", "g", "G")
    if (arg(i)%type /= ARGTYPE_REAL) then
        call msg_message (''' // char (fmt) // ''')
        call msg_error ("C-formatting string: " &
            // "argument type mismatch: real value expected")
        ok = .false.; exit SCAN_ARGS
    end if
case ("s")
    if (arg(i)%type /= ARGTYPE_STR) then
        call msg_message (''' // char (fmt) // ''')
        call msg_error ("C-formatting string: " &
            // "argument type mismatch: logical or string value expected")
        ok = .false.; exit SCAN_ARGS
    end if
case default

```

```

        call msg_message (''' // char (fmt) // ''')
        call msg_error ("C-formatting string: " &
            // "illegal or incomprehensible conversion specifier")
        ok = .false.; exit SCAN_ARGS
    end select
    split_fmt(i) = split_fmt(i) // extract (buffer, 1, 1)
    buffer = remove (buffer, 1, 1)
end do SCAN_ARGS
if (ok) then
    FIND_EXTRA_CONVERSION: do
        call split (buffer, word, "%", separator=separator)
        split_fmt(n_args) = split_fmt(n_args) // word // separator
        if (separator == "") exit FIND_EXTRA_CONVERSION
        if (extract (buffer, 1, 1) == "%") then
            split_fmt(n_args) = split_fmt(n_args) // "%"
            buffer = remove (buffer, 1, 1)
        else
            call msg_message (''' // char (fmt) // ''')
            call msg_error ("C-formatting string: " &
                // "too many conversion specifiers in format string")
            ok = .false.; exit FIND_EXTRA_CONVERSION
        end if
    end do FIND_EXTRA_CONVERSION
    split_fmt(n_args) = split_fmt(n_args) // buffer
    allocate (intf (0:n_args))
    call sprintf_interface_init (intf(0), split_fmt(0), arg_null)
    do i = 1, n_args
        call sprintf_interface_init (intf(i), split_fmt(i), arg(i))
    end do
else
    allocate (intf (0))
end if
contains
subroutine read_int_from_string (word, length, l)
    type(string_t), intent(in) :: word
    integer, intent(in) :: length
    integer, intent(out) :: l
    character(len=length) :: buffer
    buffer = word
    read (buffer, *) l
end subroutine read_int_from_string
end subroutine chop_and_check_format_string

```

## 2.12.2 API

*(Formats: public)*+≡

```
public :: sprintf
```

*(Formats: procedures)*+≡

```
function sprintf (fmt, arg) result (string)
    type(string_t) :: string
    type(string_t), intent(in) :: fmt
    type(sprintf_arg_t), dimension(:), intent(in) :: arg

```

```

type(sprintf_interface_t), dimension(:), allocatable :: intf
integer :: i
string = ""
call chop_and_check_format_string (fmt, arg, intf)
if (size (intf) > 0) then
  do i = 0, ubound (intf, 1)
    call sprintf_interface_apply_sprintf (intf(i))
    string = string // sprintf_interface_get_result (intf(i))
  end do
end if
end function sprintf

```

### 2.12.3 Test

*(Formats: public)*+≡

```
public :: format_test
```

*(Formats: procedures)*+≡

```

subroutine format_test ()
  print *, "*** 1. Test: a string ***"
  call test_run (var_str("%s"), 1, (/ 4 /), (/ 'abcdefghij' /))
  print *, "*** 2. Test: two integers ***"
  call test_run (var_str("%d,%d"), 2, (/ 2, 2 /), (/ '42', '13' /))
  print *, "*** 3. Test: floating point number ***"
  call test_run (var_str("%8.4f"), 1, (/ 3 /), (/ '42567.12345' /))
  print *, "*** 4. Test: general expression ***"
  call test_run (var_str("%g"), 1, (/ 3 /), (/ '3.1415' /))
contains
  subroutine test_run (fmt, n_args, type, buffer)
    type(string_t), intent(in) :: fmt
    integer, intent(in) :: n_args
    logical :: lval
    integer :: ival
    real(default) :: rval
    integer :: i
    type(string_t) :: string
    type(sprintf_arg_t), dimension(:), allocatable :: arg
    integer, dimension(n_args), intent(in) :: type
    character(*), dimension(n_args), intent(in) :: buffer
    print *, "Format string:", char(fmt)
    print *, "Number of args:", n_args
    allocate (arg (n_args))
    do i = 1, n_args
      print *, "Argument (type ) = ", type(i)
      select case (type(i))
      case (ARGTYPE_LOG)
        read (buffer(i), *) lval
        call sprintf_arg_init (arg(i), lval)
      case (ARGTYPE_INT)
        read (buffer(i), *) ival
        call sprintf_arg_init (arg(i), ival)
      case (ARGTYPE_REAL)
        read (buffer(i), *) rval

```

```

        call sprintf_arg_init (arg(i), rval)
    case (ARGTYPE_STR)
        call sprintf_arg_init (arg(i), var_str (trim (buffer(i))))
    end select
end do
string = sprintf (fmt, arg)
print *, "Result: '", char (string), "'"
deallocate (arg)
end subroutine test_run
end subroutine format_test

```

## 2.13 Bytes and such

In a few instances we will need the notion of a byte (8-bit) and a word (32 bit), even a 64-bit word. A block of 512 bit is also needed (for MD5).

We rely on integers up to 64 bit being supported by the processor. The main difference to standard integers is the interpretation as unsigned integers.

```

<bytes.f90>≡
  <File header>

  module bytes

    use kinds, only: i8, i32, i64 !NODEP!
    <Use file utils>

    <Standard module head>

    <Bytes: public>

    <Bytes: types>

    <Bytes: parameters>

    <Bytes: interfaces>

    contains

    <Bytes: procedures>

  end module bytes

```

### 2.13.1 8-bit words: bytes

This is essentially a wrapper around 8-bit integers. The wrapper emphasises their special interpretation as a sequence of bits. However, we interpret bytes as unsigned integers.

```

<Bytes: public>≡
  public :: byte_t

```

```

<Bytes: types>≡
    type :: byte_t
        private
            integer(i8) :: i
        end type byte_t

<Bytes: public>+≡
    public :: byte_zero

<Bytes: parameters>≡
    type(byte_t), parameter :: byte_zero = byte_t (0_i8)

```

Set a byte from 8-bit integer:

```

<Bytes: public>+≡
    public :: assignment(=)

<Bytes: interfaces>≡
    interface assignment(=)
        module procedure set_byte_from_i8
    end interface

<Bytes: procedures>≡
    subroutine set_byte_from_i8 (b, i)
        type(byte_t), intent(out) :: b
        integer(i8), intent(in) :: i
        b%i = i
    end subroutine set_byte_from_i8

```

Write a byte in one of two formats: either as a hexadecimal number (two digits, default) or as a decimal number (one to three digits). The decimal version is nontrivial because bytes are unsigned integers. Optionally append a newline.

```

<Bytes: public>+≡
    public :: byte_write

<Bytes: interfaces>+≡
    interface byte_write
        module procedure byte_write_unit, byte_write_string
    end interface

<Bytes: procedures>+≡
    subroutine byte_write_unit (b, unit, decimal, newline)
        type(byte_t), intent(in), optional :: b
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: decimal, newline
        logical :: dc, nl
        type(word32_t) :: w
        integer :: u
        u = output_unit (unit); if (u < 0) return
        dc = .false.; if (present (decimal)) dc = decimal
        nl = .false.; if (present (newline)) nl = newline
        if (dc) then
            w = b
            write (u, '(I3)', advance='no') w%i
        else
            write (u, '(z2.2)', advance='no') b%i
        end if
    end subroutine byte_write_unit

```

```

    end if
    if (nl) write (u, *)
end subroutine byte_write_unit

```

The string version is hex-only

```

<Bytes: procedures>+≡
subroutine byte_write_string (b, s)
  type(byte_t), intent(in) :: b
  character(len=2), intent(inout) :: s
  write (s, '(z2.2)') b%i
end subroutine byte_write_string

```

### 2.13.2 32-bit words

This is not exactly a 32-bit integer. A word is to be filled with bytes, and it may be partially filled. The filling is done lowest-byte first, highest-byte last. We count the bits, so `fill` should be either 0, 8, 16, 24, or 32. In printing words, we correspondingly distinguish between printing zeros and printing blanks.

```

<Bytes: public>+≡
public :: word32_t

<Bytes: types>+≡
type :: word32_t
  private
  integer(i32) :: i
  integer :: fill = 0
end type word32_t

```

Assignment: the word is filled by inserting a 32-bit integer

```

<Bytes: interfaces>+≡
interface assignment(=)
  module procedure word32_set_from_i32
  module procedure word32_set_from_byte
end interface

<Bytes: procedures>+≡
subroutine word32_set_from_i32 (w, i)
  type(word32_t), intent(out) :: w
  integer(i32), intent(in) :: i
  w%i = i
  w%fill = 32
end subroutine word32_set_from_i32

```

Reverse assignment to a 32-bit integer. We do not check the fill status.

```

<Bytes: interfaces>+≡
interface assignment(=)
  module procedure i32_from_word32
end interface

```



```

<Bytes: procedures>+≡
  subroutine i32_from_word32 (i, w)
    integer(i32), intent(out) :: i
    type(word32_t), intent(in) :: w
    i = w%i
  end subroutine i32_from_word32

```

Filling with a 8-bit integer is slightly tricky, because in this interpretation integers are unsigned.

```

<Bytes: procedures>+≡
  subroutine word32_set_from_byte (w, b)
    type(word32_t), intent(out) :: w
    type(byte_t), intent(in) :: b
    if (b%i >= 0_i8) then
      w%i = b%i
    else
      w%i = 2_i32*(huge(0_i8)+1_i32) + b%i
    end if
    w%fill = 32
  end subroutine word32_set_from_byte

```

Check the fill status

```

<Bytes: public>+≡
  public :: word32_empty, word32_filled, word32_fill

<Bytes: procedures>+≡
  function word32_empty (w)
    type(word32_t), intent(in) :: w
    logical :: word32_empty
    word32_empty = (w%fill == 0)
  end function word32_empty

  function word32_filled (w)
    type(word32_t), intent(in) :: w
    logical :: word32_filled
    word32_filled = (w%fill == 32)
  end function word32_filled

  function word32_fill (w)
    type(word32_t), intent(in) :: w
    integer :: word32_fill
    word32_fill = w%fill
  end function word32_fill

```

Partial assignment: append a byte to a partially filled word. (Note: no assignment if the word is filled, so check this before if necessary.)

```

<Bytes: public>+≡
  public :: word32_append_byte

<Bytes: procedures>+≡
  subroutine word32_append_byte (w, b)
    type(word32_t), intent(inout) :: w
    type(byte_t), intent(in) :: b

```

```

type(word32_t) :: w1
if (.not. word32_filled (w)) then
    w1 = b
    call mvbits (w1%i, 0, 8, w%i, w%fill)
    w%fill = w%fill + 8
end if
end subroutine word32_append_byte

```

Extract a byte from a word. The argument *i* is the position, which may be 0, 1, 2, or 3.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

```

<Bytes: public>+≡
    public :: byte_from_word32

<Bytes: procedures>+≡
    function byte_from_word32 (w, i) result (b)
        type(word32_t), intent(in) :: w
        integer, intent(in) :: i
        type(byte_t) :: b
        integer(i32) :: j
        j = 0
        if (i >= 0 .and. i*8 < w%fill) then
            call mvbits (w%i, i*8, 8, j, 0)
        end if
        b%i = int (ibclr (j, 7), kind=i8)
        if (btest (j, 7)) b%i = ibset (b%i, 7)
    end function byte_from_word32

```

Write a word to file or STDOUT. We understand words as unsigned integers, therefore we cannot use the built-in routine unchanged. However, we can make use of the existence of 64-bit integers and their output routine.

In hexadecimal format, the default version prints eight hex characters, highest-first. The **bytes** version prints four bytes (two-hex characters), lowest first, with spaces in-between. The decimal bytes version is analogous. In the **bytes** version, missing bytes are printed as whitespace.

```

<Bytes: public>+≡
    public :: word32_write

<Bytes: interfaces>+≡
    interface word32_write
        module procedure word32_write_unit
    end interface

<Bytes: procedures>+≡
    subroutine word32_write_unit (w, unit, bytes, decimal, newline)
        type(word32_t), intent(in) :: w
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: bytes, decimal, newline
        logical :: dc, by, nl
        type(word64_t) :: ww
        integer :: i, u

```

```

u = output_unit (unit); if (u < 0) return
by = .false.; if (present (bytes)) by = bytes
dc = .false.; if (present (decimal)) dc = decimal
nl = .false.; if (present (newline)) nl = newline
if (by) then
  do i = 0, 3
    if (i>0) write (u, '(1x)', advance='no')
    if (8*i < w%fill) then
      call byte_write (byte_from_word32 (w, i), unit, decimal=decimal)
    else if (dc) then
      write (u, '(3x)', advance='no')
    else
      write (u, '(2x)', advance='no')
    end if
  end do
else if (dc) then
  ww = w
  write (u, '(I10)', advance='no') ww%i
else
  select case (w%fill)
  case ( 0)
  case ( 8); write (6, '(1x,z8.2)', advance='no') ibits (w%i, 0, 8)
  case (16); write (6, '(1x,z8.4)', advance='no') ibits (w%i, 0,16)
  case (24); write (6, '(1x,z8.6)', advance='no') ibits (w%i, 0,24)
  case (32); write (6, '(1x,z8.8)', advance='no') ibits (w%i, 0,32)
  end select
end if
if (nl) write (u, *)
end subroutine word32_write_unit

```

### 2.13.3 Operations on 32-bit words

Define the usual logical operations, as well as addition (mod  $2^{32}$ ). We assume that all operands are completely filled.

*(Bytes: public)*+≡

```
public :: not, ior, ieor, iand, ishft, ishftc
```

*(Bytes: interfaces)*+≡

```

interface not
  module procedure word_not
end interface
interface ior
  module procedure word_or
end interface
interface ieor
  module procedure word_eor
end interface
interface iand
  module procedure word_and
end interface
interface ishft
  module procedure word_shft
end interface

```

```

interface ishftc
  module procedure word_shftc
end interface

<Bytes: procedures>+≡
function word_not (w1) result (w2)
  type(word32_t), intent(in) :: w1
  type(word32_t) :: w2
  w2 = not (w1%i)
end function word_not

function word_or (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = ior (w1%i, w2%i)
end function word_or

function word_eor (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = ieor (w1%i, w2%i)
end function word_eor

function word_and (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = iand (w1%i, w2%i)
end function word_and

function word_shft (w1, s) result (w2)
  type(word32_t), intent(in) :: w1
  integer, intent(in) :: s
  type(word32_t) :: w2
  w2 = ishft (w1%i, s)
end function word_shft

function word_shftc (w1, s) result (w2)
  type(word32_t), intent(in) :: w1
  integer, intent(in) :: s
  type(word32_t) :: w2
  w2 = ishftc (w1%i, s, 32)
end function word_shftc

```

Addition is defined mod  $2^{32}$ , i.e., without overflow checking. This means that we have to work around a possible overflow check enforced by the compiler.

```

<Bytes: public>+≡
public :: operator(+)

<Bytes: interfaces>+≡
interface operator(+)
  module procedure word_add
  module procedure word_add_i8
  module procedure word_add_i32
end interface

```

```

<Bytes: procedures>+≡
function word_add (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  integer(i64) :: j
  j = int (ibclr (w1%i, 31), i64) + int (ibclr (w2%i, 31), i64)
  w3 = int (ibclr (j, 31), kind=i32)
  if (btest (j, 31)) then
    if (btest (w1%i, 31) .eqv. btest (w2%i, 31)) w3 = ibset (w3%i, 31)
  else
    if (btest (w1%i, 31) .neqv. btest (w2%i, 31)) w3 = ibset (w3%i, 31)
  end if
end function word_add

function word_add_i8 (w1, i) result (w3)
  type(word32_t), intent(in) :: w1
  integer(i8), intent(in) :: i
  type(word32_t) :: w3
  integer(i64) :: j
  j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 7), i64)
  if (btest (i, 7)) j = j + 128
  w3 = int (ibclr (j, 31), kind=i32)
  if (btest (j, 31) .neqv. btest (w1%i, 31)) w3 = ibset (w3%i, 31)
end function word_add_i8

function word_add_i32 (w1, i) result (w3)
  type(word32_t), intent(in) :: w1
  integer(i32), intent(in) :: i
  type(word32_t) :: w3
  integer(i64) :: j
  j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 31), i64)
  w3 = int (ibclr (j, 31), kind=i32)
  if (btest (j, 31)) then
    if (btest (w1%i, 31) .eqv. btest (i, 31)) w3 = ibset (w3%i, 31)
  else
    if (btest (w1%i, 31) .neqv. btest (i, 31)) w3 = ibset (w3%i, 31)
  end if
end function word_add_i32

```

#### 2.13.4 64-bit words

These objects consist of two 32-bit words. They thus can hold integer numbers larger than  $2^{32}$  (to be exact,  $2^{31}$  since FORTRAN integers are signed). The order is low-word, high-word.

```

<Bytes: public>+≡
public :: word64_t

<Bytes: types>+≡
type :: word64_t
  private
  integer(i64) :: i
end type word64_t

```

Set a 64 bit word:

```

<Bytes: interfaces>+≡
    interface assignment(=)
        module procedure word64_set_from_i64
        module procedure word64_set_from_word32
    end interface

<Bytes: procedures>+≡
    subroutine word64_set_from_i64 (ww, i)
        type(word64_t), intent(out) :: ww
        integer(i64), intent(in) :: i
        ww%i = i
    end subroutine word64_set_from_i64

```

Filling with a 32-bit word:

```

<Bytes: procedures>+≡
    subroutine word64_set_from_word32 (ww, w)
        type(word64_t), intent(out) :: ww
        type(word32_t), intent(in) :: w
        if (w%i >= 0_i32) then
            ww%i = w%i
        else
            ww%i = 2_i64*(huge(0_i32)+1_i64) + w%i
        end if
    end subroutine word64_set_from_word32

```

Extract a byte from a word. The argument *i* is the position, which may be between 0 and 7.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

```

<Bytes: public>+≡
    public :: byte_from_word64, word32_from_word64

<Bytes: procedures>+≡
    function byte_from_word64 (ww, i) result (b)
        type(word64_t), intent(in) :: ww
        integer, intent(in) :: i
        type(byte_t) :: b
        integer(i64) :: j
        j = 0
        if (i >= 0 .and. i*8 < 64) then
            call mvbits (ww%i, i*8, 8, j, 0)
        end if
        b%i = int (ibclr (j, 7), kind=i8)
        if (btest (j, 7)) b%i = ibset (b%i, 7)
    end function byte_from_word64

```

Extract a 32-bit word from a 64-bit word. The position is either 0 or 1.

```

<Bytes: procedures>+≡
    function word32_from_word64 (ww, i) result (w)
        type(word64_t), intent(in) :: ww

```

```

integer, intent(in) :: i
type(word32_t) :: w
integer(i64) :: j
j = 0
select case (i)
case (0); call mvbits (ww%i, 0, 32, j, 0)
case (1); call mvbits (ww%i, 32, 32, j, 0)
end select
w = int (ibclr (j, 31), kind=i32)
if (btest (j, 31)) w = ibset (w%i, 31)
end function word32_from_word64

```

Print a 64-bit word. Decimal version works up to  $2^{63}$ . The `words` version uses the 'word32' printout, separated by two spaces. The low-word is printed first. The `bytes` version also uses the 'word32' printout. This implies that the lowest byte is first. The default version prints a hexadecimal number without spaces, highest byte first.

*(Bytes: public)+≡*

```
public :: word64_write
```

*(Bytes: interfaces)+≡*

```

interface word64_write
  module procedure word64_write_unit
end interface

```

*(Bytes: procedures)+≡*

```

subroutine word64_write_unit (ww, unit, words, bytes, decimal, newline)
  type(word64_t), intent(in) :: ww
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: words, bytes, decimal, newline
  logical :: wo, by, dc, nl
  integer :: u
  u = output_unit (unit); if (u < 0) return
  wo = .false.; if (present (words)) wo = words
  by = .false.; if (present (bytes)) by = bytes
  dc = .false.; if (present (decimal)) dc = decimal
  nl = .false.; if (present (newline)) nl = newline
  if (wo .or. by) then
    call word32_write_unit (word32_from_word64 (ww, 0), unit, by, dc)
    write (u, '(2x)', advance='no')
    call word32_write_unit (word32_from_word64 (ww, 1), unit, by, dc)
  else if (dc) then
    write (u, '(I19)', advance='no') ww%i
  else
    write (u, '(Z16)', advance='no') ww%i
  end if
  if (nl) write (u, *)
end subroutine word64_write_unit

```

## 2.14 MD5 Checksums

It is a bit of an overkill, but implementing MD5 checksums allows us to check input/file integrity on the basis of a well-known standard. The building blocks have been introduced in the `bytes` module.

```
<md5.f90>≡  
<File header>  
  
module md5  
  
    use kinds, only: i8, i32, i64 !NODEP!  
<Use file utils>  
    use diagnostics !NODEP!  
    use bytes  
    use limits, only: LF, EOR, EOF !NODEP!  
  
<Standard module head>  
  
<MD5: public>  
  
<MD5: types>  
  
<MD5: variables>  
  
<MD5: interfaces>  
  
contains  
  
<MD5: procedures>  
  
end module md5
```

### 2.14.1 Blocks

A block is a sequence of 16 words (64 bytes or 512 bits). We anticipate that blocks will be linked, so include a pointer to the next block. There is a fill status (word counter), as there is one for each word. The fill status is equal to the number of bytes that are in, so it may be between 0 and 64.

```
<MD5: types>≡  
    type :: block_t  
        private  
        type(word32_t), dimension(0:15) :: w  
        type(block_t), pointer :: next => null ()  
        integer :: fill = 0  
    end type block_t
```

Check if a block is completely filled or empty:

```
<MD5: procedures>≡  
    function block_is_empty (b)  
        type(block_t), intent(in) :: b  
        logical :: block_is_empty  
        block_is_empty = (b%fill == 0 .and. word32_empty (b%w(0)))
```



```

end function block_is_empty

function block_is_filled (b)
  type(block_t), intent(in) :: b
  logical :: block_is_filled
  block_is_filled = (b%fill == 64)
end function block_is_filled

```

Append a single byte to a block. Works only if the block is not yet filled.

*(MD5: procedures)+≡*

```

subroutine block_append_byte (bl, by)
  type(block_t), intent(inout) :: bl
  type(byte_t), intent(in) :: by
  if (.not. block_is_filled (bl)) then
    call word32_append_byte (bl%w(bl%fill/4), by)
    bl%fill = bl%fill + 1
  end if
end subroutine block_append_byte

```

The printing routine allows for printing as sequences of words or bytes, decimal or hex.

*(MD5: interfaces)≡*

```

interface block_write
  module procedure block_write_unit
end interface

```

*(MD5: procedures)+≡*

```

subroutine block_write_unit (b, unit, bytes, decimal)
  type(block_t), intent(in) :: b
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  logical :: by, dc
  integer :: i, u
  u = output_unit (unit); if (u < 0) return
  by = .false.; if (present (bytes)) by = bytes
  dc = .false.; if (present (decimal)) dc = decimal
  do i = 0, b%fill/4 - 1
    call newline_or_blank (u, i, by, dc)
    call word32_write (b%w(i), unit, bytes, decimal)
  end do
  if (.not. block_is_filled (b)) then
    i = b%fill/4
    if (.not. word32_empty (b%w(i))) then
      call newline_or_blank (u, i, by, dc)
      call word32_write (b%w(i), unit, bytes, decimal)
    end if
  end if
  write (u, *)
contains
  subroutine newline_or_blank (u, i, bytes, decimal)
    integer, intent(in) :: u, i
    logical, intent(in) :: bytes, decimal
    if (decimal) then

```

```

        select case (i)
        case (0)
        case (2,4,6,8,10,12,14); write (u, *)
        case default
            write (u, '(2x)', advance='no')
        end select
    else if (bytes) then
        select case (i)
        case (0)
        case (4,8,12); write (u, *)
        case default
            write (u, '(2x)', advance='no')
        end select
    else
        if (i == 8) write (u, *)
    end if
end subroutine newline_or_blank
end subroutine block_write_unit

```

### 2.14.2 Messages

A message (within this module) is a linked list of blocks.

*(MD5: types)+≡*

```

type :: message_t
private
type(block_t), pointer :: first => null ()
type(block_t), pointer :: last => null ()
integer :: n_blocks = 0
end type message_t

```

Clear the message list

*(MD5: procedures)+≡*

```

subroutine message_clear (m)
type(message_t), intent(inout) :: m
type(block_t), pointer :: b
nullify (m%last)
do
    b => m%first
    if (.not.(associated (b))) exit
    m%first => b%next
    deallocate (b)
end do
m%n_blocks = 0
end subroutine message_clear

```

Append an empty block to the message list

*(MD5: procedures)+≡*

```

subroutine message_append_new_block (m)
type(message_t), intent(inout) :: m
if (associated (m%last)) then
    allocate (m%last%next)

```

```

        m%last => m%last%next
        m%n_blocks = m%n_blocks + 1
    else
        allocate (m%first)
        m%last => m%first
        m%n_blocks = 1
    end if
end subroutine message_append_new_block

```

Initialize: clear and allocate the first (empty) block.

```

<MD5: procedures>+≡
subroutine message_init (m)
    type(message_t), intent(inout) :: m
    call message_clear (m)
    call message_append_new_block (m)
end subroutine message_init

```

Append a single byte to a message. If necessary, allocate a new block. If the message is empty, initialize it.

```

<MD5: procedures>+≡
subroutine message_append_byte (m, b)
    type(message_t), intent(inout) :: m
    type(byte_t), intent(in) :: b
    if (.not. associated (m%last)) then
        call message_init (m)
    else if (block_is_filled (m%last)) then
        call message_append_new_block (m)
    end if
    call block_append_byte (m%last, b)
end subroutine message_append_byte

```

Append zero bytes until the current block is filled up to the required position. If we are already beyond that, append a new block and fill that one.

```

<MD5: procedures>+≡
subroutine message_pad_zero (m, i)
    type(message_t), intent(inout) :: m
    integer, intent(in) :: i
    type(block_t), pointer :: b
    integer :: j
    if (associated (m%last)) then
        b => m%last
        if (b%fill > i) then
            do j = b%fill + 1, 64 + i
                call message_append_byte (m, byte_zero)
            end do
        else
            do j = b%fill + 1, i
                call message_append_byte (m, byte_zero)
            end do
        end if
    end if
end subroutine message_pad_zero

```

This returns the number of bits within a message. We need a 64-bit word for the result since it may be more than  $2^{31}$ . This is also required by the MD5 standard.

*(MD5: procedures)+≡*

```
function message_bits (m) result (length)
  type(message_t), intent(in) :: m
  type(word64_t) :: length
  type(block_t), pointer :: b
  integer(i64) :: n_blocks_filled, n_bytes_extra
  if (m%n_blocks > 0) then
    b => m%last
    if (block_is_filled (b)) then
      n_blocks_filled = m%n_blocks
      n_bytes_extra = 0
    else
      n_blocks_filled = m%n_blocks - 1
      n_bytes_extra = b%fill
    end if
    length = n_blocks_filled * 512 + n_bytes_extra * 8
  else
    length = 0_i64
  end if
end function message_bits
```

### 2.14.3 Message I/O

Append the contents of a string to a message. We first cast the character string into a 8-bit integer array and then append this byte by byte.

*(MD5: procedures)+≡*

```
subroutine message_append_string (m, s)
  type(message_t), intent(inout) :: m
  character(len=*), intent(in) :: s
  integer(i64) :: i, n_bytes
  integer(i8), dimension(:), allocatable :: buffer
  integer(i8), dimension(1) :: mold
  type(byte_t) :: b
  n_bytes = size (transfer (s, mold))
  allocate (buffer (n_bytes))
  buffer = transfer (s, mold)
  do i = 1, size (buffer)
    b = buffer(i)
    call message_append_byte (m, b)
  end do
  deallocate (buffer)
end subroutine message_append_string
```

Append the contents of a 32-bit integer to a message. We first cast the 32-bit integer into a 8-bit integer array and then append this byte by byte.

*(MD5: procedures)+≡*

```
subroutine message_append_i32 (m, x)
```

```

type(message_t), intent(inout) :: m
integer(i32), intent(in) :: x
integer(i8), dimension(4) :: buffer
type(byte_t) :: b
integer :: i
buffer = transfer (x, buffer, size(buffer))
do i = 1, size (buffer)
    b = buffer(i)
    call message_append_byte (m, b)
end do
end subroutine message_append_i32

```

Append one line from file to a message. Include the newline character.

*(MD5: procedures)+≡*

```

! subroutine message_append_from_unit (m, u, iostat)
!   type(message_t), intent(inout) :: m
!   integer, intent(in) :: u
!   integer, intent(out) :: iostat
!   character(len=BUFFER_SIZE) :: buffer
!   read (u, *, iostat=iostat) buffer
!   call message_append_string (m, trim (buffer))
!   call message_append_string (m, LF)
! end subroutine message_append_from_unit

```

Fill a message from file. (Each line counts as a string.)

*(MD5: procedures)+≡*

```

! subroutine message_read_from_file (m, f)
!   type(message_t), intent(inout) :: m
!   character(len=*), intent(in) :: f
!   integer :: u, iostat
!   u = free_unit ()
!   open (file=f, unit=u, action='read')
!   do
!       call message_append_from_unit (m, u, iostat=iostat)
!       if (iostat < 0) exit
!   end do
!   close (u)
! end subroutine message_read_from_file

```

Write a message. After each block, insert an empty line.

*(MD5: interfaces)+≡*

```

interface message_write
    module procedure message_write_unit
end interface

```

*(MD5: procedures)+≡*

```

subroutine message_write_unit (m, unit, bytes, decimal)
    type(message_t), intent(in) :: m
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: bytes, decimal
    type(block_t), pointer :: b
    integer :: u
    u = output_unit (unit); if (u < 0) return

```

```

b => m%first
if (associated (b)) then
  do
    call block_write_unit (b, unit, bytes, decimal)
    b => b%next
    if (.not. associated (b)) exit
    write (u, *)
  end do
end if
end subroutine message_write_unit

```

#### 2.14.4 Auxiliary functions

These four functions on three words are defined in the MD5 standard:

```

⟨MD5: procedures⟩+≡
function ff (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: ff
  ff = ior (iand (x, y), iand (not (x), z))
end function ff

function fg (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fg
  fg = ior (iand (x, z), iand (y, not (z)))
end function fg

function fh (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fh
  fh = ieor (ieor (x, y), z)
end function fh

function fi (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fi
  fi = ieor (y, ior (x, not (z)))
end function fi

```

#### 2.14.5 Auxiliary stuff

This defines and initializes the table of transformation constants:

```

⟨MD5: variables⟩≡
  type(word32_t), dimension(64), save :: t
  logical, save :: table_initialized = .false.

⟨MD5: procedures⟩+≡
  subroutine table_init
    type(word64_t) :: ww
    integer :: i
    if (.not.table_initialized) then

```

```

do i = 1, 64
  ww = int (4294967296d0 * abs (sin (i * 1d0)), kind=i64)
  t(i) = word32_from_word64 (ww, 0)
end do
table_initialized = .true.
end if
end subroutine table_init

```

This encodes the message digest (4 words) into a 32-character string.

*(MD5: procedures)+≡*

```

function digest_string (aa) result (s)
  type(word32_t), dimension (0:3), intent(in) :: aa
  character(len=32) :: s
  integer :: i, j
  do i = 0, 3
    do j = 0, 3
      call byte_write (byte_from_word32 (aa(i), j), s(i*8+j*2+1:i*8+j*2+2))
    end do
  end do
end function digest_string

```

## 2.14.6 MD5 algorithm

Pad the message with a byte x80 and then pad zeros up to a full block minus two words; in these words, insert the message length (before padding) as a 64-bit word, low-word first.

*(MD5: procedures)+≡*

```

subroutine message_pad (m)
  type(message_t), intent(inout) :: m
  type(word64_t) :: length
  integer(i8), parameter :: ipad = -128 ! z'80'
  type(byte_t) :: b
  integer :: i
  length = message_bits (m)
  b = ipad
  call message_append_byte (m, b)
  call message_pad_zero (m, 56)
  do i = 0, 7
    call message_append_byte (m, byte_from_word64 (length, i))
  end do
end subroutine message_pad

```

Apply a series of transformations onto a state a,b,c,d, where the transform function uses each word of the message together with the predefined words. Finally, encode the state as a 32-character string.

*(MD5: procedures)+≡*

```

subroutine message_digest (m, s)
  type(message_t), intent(in) :: m
  character(len=32), intent(out) :: s
  integer(i32), parameter :: ia = 1732584193 ! z'67452301'
  integer(i32), parameter :: ib = -271733879 ! z'efcdab89'

```

```

integer(i32), parameter :: ic = -1732584194 ! z'98badcfe'
integer(i32), parameter :: id = 271733878 ! z'10325476'
type(word32_t) :: a, b, c, d
type(word32_t) :: aa, bb, cc, dd
type(word32_t), dimension(0:15) :: x
type(block_t), pointer :: bl
call table_init
a = ia; b = ib; c = ic; d = id
bl => m%first
do
  if (.not.associated (bl)) exit
  x = bl%w
  aa = a; bb = b; cc = c; dd = d
  call transform (ff, a, b, c, d, 0, 7, 1)
  call transform (ff, d, a, b, c, 1, 12, 2)
  call transform (ff, c, d, a, b, 2, 17, 3)
  call transform (ff, b, c, d, a, 3, 22, 4)
  call transform (ff, a, b, c, d, 4, 7, 5)
  call transform (ff, d, a, b, c, 5, 12, 6)
  call transform (ff, c, d, a, b, 6, 17, 7)
  call transform (ff, b, c, d, a, 7, 22, 8)
  call transform (ff, a, b, c, d, 8, 7, 9)
  call transform (ff, d, a, b, c, 9, 12, 10)
  call transform (ff, c, d, a, b, 10, 17, 11)
  call transform (ff, b, c, d, a, 11, 22, 12)
  call transform (ff, a, b, c, d, 12, 7, 13)
  call transform (ff, d, a, b, c, 13, 12, 14)
  call transform (ff, c, d, a, b, 14, 17, 15)
  call transform (ff, b, c, d, a, 15, 22, 16)
  call transform (fg, a, b, c, d, 1, 5, 17)
  call transform (fg, d, a, b, c, 6, 9, 18)
  call transform (fg, c, d, a, b, 11, 14, 19)
  call transform (fg, b, c, d, a, 0, 20, 20)
  call transform (fg, a, b, c, d, 5, 5, 21)
  call transform (fg, d, a, b, c, 10, 9, 22)
  call transform (fg, c, d, a, b, 15, 14, 23)
  call transform (fg, b, c, d, a, 4, 20, 24)
  call transform (fg, a, b, c, d, 9, 5, 25)
  call transform (fg, d, a, b, c, 14, 9, 26)
  call transform (fg, c, d, a, b, 3, 14, 27)
  call transform (fg, b, c, d, a, 8, 20, 28)
  call transform (fg, a, b, c, d, 13, 5, 29)
  call transform (fg, d, a, b, c, 2, 9, 30)
  call transform (fg, c, d, a, b, 7, 14, 31)
  call transform (fg, b, c, d, a, 12, 20, 32)
  call transform (fh, a, b, c, d, 5, 4, 33)
  call transform (fh, d, a, b, c, 8, 11, 34)
  call transform (fh, c, d, a, b, 11, 16, 35)
  call transform (fh, b, c, d, a, 14, 23, 36)
  call transform (fh, a, b, c, d, 1, 4, 37)
  call transform (fh, d, a, b, c, 4, 11, 38)
  call transform (fh, c, d, a, b, 7, 16, 39)
  call transform (fh, b, c, d, a, 10, 23, 40)
  call transform (fh, a, b, c, d, 13, 4, 41)

```



```

call transform (fh, d, a, b, c, 0, 11, 42)
call transform (fh, c, d, a, b, 3, 16, 43)
call transform (fh, b, c, d, a, 6, 23, 44)
call transform (fh, a, b, c, d, 9, 4, 45)
call transform (fh, d, a, b, c, 12, 11, 46)
call transform (fh, c, d, a, b, 15, 16, 47)
call transform (fh, b, c, d, a, 2, 23, 48)
call transform (fi, a, b, c, d, 0, 6, 49)
call transform (fi, d, a, b, c, 7, 10, 50)
call transform (fi, c, d, a, b, 14, 15, 51)
call transform (fi, b, c, d, a, 5, 21, 52)
call transform (fi, a, b, c, d, 12, 6, 53)
call transform (fi, d, a, b, c, 3, 10, 54)
call transform (fi, c, d, a, b, 10, 15, 55)
call transform (fi, b, c, d, a, 1, 21, 56)
call transform (fi, a, b, c, d, 8, 6, 57)
call transform (fi, d, a, b, c, 15, 10, 58)
call transform (fi, c, d, a, b, 6, 15, 59)
call transform (fi, b, c, d, a, 13, 21, 60)
call transform (fi, a, b, c, d, 4, 6, 61)
call transform (fi, d, a, b, c, 11, 10, 62)
call transform (fi, c, d, a, b, 2, 15, 63)
call transform (fi, b, c, d, a, 9, 21, 64)
a = a + aa
b = b + bb
c = c + cc
d = d + dd
bl => bl%next
end do
s = digest_string ((/a, b, c, d/))
contains
<MD5: Internal subroutine transform>
end subroutine message_digest

```

And this is the actual transformation that depends on one of the previous functions, four words, and three integers. The implicit arguments are *x*, the word from the message to digest, and *t*, the entry in the predefined table.

```

<MD5: Internal subroutine transform>≡
subroutine transform (f, a, b, c, d, k, s, i)
interface
function f (x, y, z)
import word32_t
type(word32_t), intent(in) :: x, y, z
type(word32_t) :: f
end function f
end interface
type(word32_t), intent(inout) :: a
type(word32_t), intent(in) :: b, c, d
integer, intent(in) :: k, s, i
a = b + ishftc (a + f(b, c, d) + x(k) + t(i), s)
end subroutine transform

```

### 2.14.7 User interface

```
<MD5: public>≡  
public :: md5sum
```

```
<MD5: interfaces>+≡  
interface md5sum  
  module procedure md5sum_from_string  
  module procedure md5sum_from_unit  
end interface
```

This function computes the MD5 sum of the input string and returns it as a 32-character string

```
<MD5: procedures>+≡  
function md5sum_from_string (s) result (digest)  
  character(len=*), intent(in) :: s  
  character(len=32) :: digest  
  type(message_t) :: m  
  call message_append_string (m, s)  
  call message_pad (m)  
  call message_digest (m, digest)  
  call message_clear (m)  
end function md5sum_from_string
```

This funct. reads from unit u (an unformatted sequence of integers) and computes the MD5 sum.

```
<MD5: procedures>+≡  
function md5sum_from_unit (u) result (digest)  
  integer, intent(in) :: u  
  character(len=32) :: digest  
  type(message_t) :: m  
  character :: char  
  integer :: iostat  
  READ_CHARS: do  
    read (u, "(A)", advance="no", iostat=iostat) char  
    select case (iostat)  
    case (0)  
      call message_append_string (m, char)  
    case (EOR)  
      call message_append_string (m, LF)  
    case (EOF)  
      exit READ_CHARS  
    case default  
      call msg_fatal &  
        ("Computing MD5 sum: I/O error while reading from scratch file")  
    end select  
  end do READ_CHARS  
  call message_pad (m)  
  call message_digest (m, digest)  
  call message_clear (m)  
end function md5sum_from_unit
```

This funct checks the implementation by computing the checksum of certain strings and comparing them with the known values. If some inconsistency is

$\langle MD5: public \rangle + \equiv$ 
$$\langle MD5: procedures \rangle + \equiv$$

```
end subroutine md5_test
```

 $\langle \text{permutations.f90} \rangle \equiv$ 

```
use kinds, only: TC !NODEP!
```

 $\langle \textit{Standard module head} \rangle$ 

112

```

    <Permutations: types>

    <Permutations: interfaces>

contains

    <Permutations: procedures>

end module permutations

```

### 2.15.1 Permutations

A permutation is an array of integers. Each integer between one and `size` should occur exactly once.

```

<Permutations: public>≡
    public :: permutation_t

<Permutations: types>≡
    type :: permutation_t
    private
        integer, dimension(:), allocatable :: p
    end type permutation_t

```

Initialize with the identity permutation.

```

<Permutations: public>+≡
    public :: permutation_init
    public :: permutation_final

<Permutations: procedures>≡
    elemental subroutine permutation_init (p, size)
        type(permutation_t), intent(inout) :: p
        integer, intent(in) :: size
        integer :: i
        allocate (p%p (size))
        forall (i = 1:size)
            p%p(i) = i
        end forall
    end subroutine permutation_init

    elemental subroutine permutation_final (p)
        type(permutation_t), intent(inout) :: p
        deallocate (p%p)
    end subroutine permutation_final

```

I/O:

```

<Permutations: public>+≡
    public :: permutation_write

<Permutations: procedures>+≡
    subroutine permutation_write (p, u)
        type(permutation_t), intent (in) :: p
        integer, intent(in) :: u
        integer :: i

```

```

do i = 1, size (p%p)
  if (size (p%p) < 10) then
    write (u,"(1x,I1)", advance="no") p%p(i)
  else
    write (u,"(1x,I3)", advance="no") p%p(i)
  end if
end do
write (u, *)
end subroutine permutation_write

```

Administration:

```

⟨Permutations: public⟩+=
  public :: permutation_size

⟨Permutations: procedures⟩+=
  elemental function permutation_size (perm) result (s)
    type(permutation_t), intent(in) :: perm
    integer :: s
    s = size (perm%p)
  end function permutation_size

```

Extract an entry in a permutation.

```

⟨Permutations: public⟩+=
  public :: permute

⟨Permutations: procedures⟩+=
  elemental function permute (i, p) result (j)
    integer, intent(in) :: i
    type(permutation_t), intent(in) :: p
    integer :: j
    if (i > 0 .and. i <= size (p%p)) then
      j = p%p(i)
    else
      j = 0
    end if
  end function permute

```

Check whether a permutation is valid: Each integer in the range occurs exactly once.

```

⟨Permutations: public⟩+=
  public :: permutation_ok

⟨Permutations: procedures⟩+=
  elemental function permutation_ok (perm) result (ok)
    type(permutation_t), intent(in) :: perm
    logical :: ok
    integer :: i
    logical, dimension(:), allocatable :: set
    ok = .true.
    allocate (set (size (perm%p)))
    set = .false.
    do i = 1, size (perm%p)
      ok = (perm%p(i) > 0 .and. perm%p(i) <= size (perm%p))
    end do
  end function permutation_ok

```

```

        if (.not.ok) return
        set(perm%p(i)) = .true.
    end do
    ok = all (set)
end function permutation_ok

```

Find the permutation that transforms the second array into the first one. We assume that this is possible and unique and all bounds are set correctly.

This cannot be elemental.

```

<Permutations: public>+≡
    public :: permutation_find

<Permutations: procedures>+≡
    subroutine permutation_find (perm, a1, a2)
        type(permutation_t), intent(inout) :: perm
        integer, dimension(:), intent(in) :: a1, a2
        integer :: i, j
        if (allocated (perm%p)) deallocate (perm%p)
        allocate (perm%p (size (a1)))
        do i = 1, size (a1)
            do j = 1, size (a2)
                if (a1(i) == a2(j)) then
                    perm%p(i) = j
                    exit
                end if
            end do
            perm%p(i) = 0
        end do
    end subroutine permutation_find

```

Find all permutations that transform an array of integers into itself. The resulting permutation list is allocated with the correct length and filled.

The first step is to count the number of different entries in `code`. Next, we scan `code` again and assign a mask to each different entry, true for all identical entries. Finally, we recursively permute the identity for each possible mask.

The permutation is done as follows: A list of all permutations of the initial one with respect to the current mask is generated, then the permutations are generated in turn for each permutation in this list with the next mask. The result is always stored back into the main list, starting from the end of the current list.

```

<Permutations: public>+≡
    public :: permutation_array_make

<Permutations: procedures>+≡
    subroutine permutation_array_make (pa, code)
        type(permutation_t), dimension(:), allocatable, intent(out) :: pa
        integer, dimension(:), intent(in) :: code
        logical, dimension(size(code)) :: mask
        logical, dimension(:,:), allocatable :: imask
        integer, dimension(:), allocatable :: n_i
        type(permutation_t) :: p_init
        type(permutation_t), dimension(:), allocatable :: p_tmp
        integer :: psize, i, j, k, n_different, n, nn_k

```

```

psize = size (code)
mask = .true.
n_different = 0
do i=1, psize
  if (mask(i)) then
    n_different = n_different + 1
    mask = mask .and. (code /= code(i))
  end if
end do
allocate (imask(psize, n_different), n_i(n_different))
mask = .true.
k = 0
do i=1, psize
  if (mask(i)) then
    k = k + 1
    imask(:,k) = (code == code(i))
    n_i(k) = factorial (count(imask(:,k)))
    mask = mask .and. (code /= code(i))
  end if
end do
n = product (n_i)
allocate (pa (n))
call permutation_init (p_init, psize)
pa(1) = p_init
nn_k = 1
do k = 1, n_different
  allocate (p_tmp (n_i(k)))
  do i = nn_k, 1, -1
    call permutation_array_with_mask (p_tmp, imask(:,k), pa(i))
    do j = n_i(k), 1, -1
      pa((i-1)*n_i(k) + j) = p_tmp(j)
    end do
  end do
  deallocate (p_tmp)
  nn_k = nn_k * n_i(k)
end do
call permutation_final (p_init)
deallocate (imask, n_i)
end subroutine permutation_array_make

```

Make a list of permutations of the elements marked true in the `mask` array. The final permutation list must be allocated with the correct length ( $n!$ ). The third argument is the initial permutation to start with, which must have the same length as the `mask` array (this is not checked).

*(Permutations: procedures)* +=

```

subroutine permutation_array_with_mask (pa, mask, p_init)
  type(permutation_t), dimension(:), intent(inout) :: pa
  logical, dimension(:), intent(in) :: mask
  type(permutation_t), intent(in) :: p_init
  integer :: plen
  integer :: i, ii, j, fac_i, k, x
  integer, dimension(:), allocatable :: index
  plen = size (pa)

```

```

allocate (index(count(mask)))
ii = 0
do i = 1, size (mask)
  if (mask(i)) then
    ii = ii + 1
    index(ii) = i
  end if
end do
pa = p_init
ii = 0
fac_i = 1
do i = 1, size (mask)
  if (mask(i)) then
    ii = ii + 1
    fac_i = fac_i * ii
    x = permute (i, p_init)
    do j = 1, plen
      k = ii - mod (((j-1)*fac_i)/plen, ii)
      call insert (pa(j), x, k, ii, index)
    end do
  end if
end do
deallocate (index)
contains
subroutine insert (p, x, k, n, index)
  type(permutation_t), intent(inout) :: p
  integer, intent(in) :: x, k, n
  integer, dimension(:), intent(in) :: index
  integer :: i
  do i = n, k+1, -1
    p%p(index(i)) = p%p(index(i-1))
  end do
  p%p(index(k)) = x
end subroutine insert
end subroutine permutation_array_with_mask

```

The factorial function is needed for pre-determining the number of permutations that will be generated:

```

(Permutations: procedures) +=
function factorial (n) result (f)
  integer, intent(in) :: n
  integer :: f
  integer :: i
  f = 1
  do i=2, abs(n)
    f = f*i
  end do
end function factorial

```



## 2.15.2 Operations on binary codes

Binary codes are needed for phase-space trees. Since the permutation function uses permutations, and no other special type is involved, we put the functions here.

This is needed for phase space trees: permute bits in a tree binary code. If no permutation is given, leave as is. (We may want to access the permutation directly here if this is efficiency-critical.)

```

<Permutations: public>+=
    public :: tc_permute

<Permutations: procedures>+=
    function tc_permute (k, perm, mask_in) result (pk)
        integer(TC), intent(in) :: k, mask_in
        type(permutation_t), intent(in) :: perm
        integer(TC) :: pk
        integer :: i
        pk = iand (k, mask_in)
        do i = 1, size (perm%p)
!           if (btest(k,i-1)) pk = ibset (pk, permute (perm,i) - 1)
            if (btest(k,i-1)) pk = ibset (pk, perm%p(i)-1)
        end do
    end function tc_permute

```

This routine returns the number of set bits in the tree code value k. Hence, it is the number of externals connected to the current line. If *mask* is present, the complement of the tree code is also considered, and the smaller number is returned. This gives the true distance from the external states, taking into account the initial particles. The complement number is increased by one, since for a scattering diagram the vertex with the sum of all final-state codes is still one point apart from the initial particles.

```

<Permutations: public>+=
    public :: tc_decay_level

<Permutations: interfaces>=
    interface tc_decay_level
        module procedure decay_level_simple
        module procedure decay_level_complement
    end interface

<Permutations: procedures>+=
    function decay_level_complement (k, mask) result (l)
        integer(TC), intent(in) :: k, mask
        integer :: l
        l = min (decay_level_simple (k), &
            & decay_level_simple (ieor (k, mask)) + 1)
    end function decay_level_complement

    function decay_level_simple (k) result(l)
        integer(TC), intent(in) :: k
        integer :: l
        integer :: i
        l = 0
        do i=0, bit_size(k)-1

```

```

        if (btest(k,i)) l = l+1
    end do
end function decay_level_simple

```

## 2.16 Sorting

This small module provides functions for sorting integer or real arrays.

*<sorting.f90>*≡  
*<File header>*

```

module sorting

```

*<Use kinds>*  
 use diagnostics !NODEP!

*<Standard module head>*

*<Sorting: public>*

*<Sorting: interfaces>*

```

contains

```

*<Sorting: procedures>*

```

end module sorting

```

### 2.16.1 Implementation

The **sort** function returns, for a given integer or real array, the array sorted by increasing value. The current implementation is *mergesort*, which has  $O(n \ln n)$  behavior in all cases, and is stable for elements of equal value.

*<Sorting: public>*≡

```

    public :: sort

```

*<Sorting: interfaces>*≡

```

    interface sort
        module procedure sort_int
        module procedure sort_real
    end interface

```

The body is identical, just the interface differs.

*<Sorting: procedures>*≡

```

    function sort_int (val_in) result (val)
        integer, dimension(:), intent(in) :: val_in
        integer, dimension(size(val_in)) :: val
    <Sorting: sort>
    end function sort_int

```

```

    function sort_real (val_in) result (val)

```

```

    real(default), dimension(:), intent(in) :: val_in
    real(default), dimension(size(val_in)) :: val
    <Sorting: sort>
end function sort_real

```

```

<Sorting: sort>≡
    val = val_in( order (val_in) )

```

The `order` function returns, for a given integer or real array, the array of indices of the elements sorted by increasing value.

```

<Sorting: public>+≡
    public :: order

<Sorting: interfaces>+≡
    interface order
        module procedure order_int
        module procedure order_real
    end interface

```

```

<Sorting: procedures>+≡
    function order_int (val) result (idx)
        integer, dimension(:), intent(in) :: val
        integer, dimension(size(val)) :: idx
    <Sorting: order>
end function order_int

    function order_real (val) result (idx)
        real(default), dimension(:), intent(in) :: val
        integer, dimension(size(val)) :: idx
    <Sorting: order>
end function order_real

```

We start by individual elements, merge them to pairs, merge those to four-element subarrays, and so on. The last subarray can extend only up to the original array bound, of course, and the second of the subarrays to merge should contain at least one element.

```

<Sorting: order>≡
    integer :: n, i, s, b1, b2, e1, e2
    n = size (idx)
    forall (i = 1:n)
        idx(i) = i
    end forall
    s = 1
    do while (s < n)
        do b1 = 1, n-s, 2*s
            b2 = b1 + s
            e1 = b2 - 1
            e2 = min (e1 + s, n)
            call merge (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
        end do
        s = 2 * s
    end do

```

The merging step does the actual sorting. We take two sorted array sections and merge them to a sorted result array. We are working on the indices, and comparing is done by taking the associated `val` which is real or integer.

```

<Sorting: interfaces>+≡
  interface merge
    module procedure merge_int
    module procedure merge_real
  end interface

<Sorting: procedures>+≡
  subroutine merge_int (res, src1, src2, val)
    integer, dimension(:), intent(out) :: res
    integer, dimension(:), intent(in) :: src1, src2
    integer, dimension(:), intent(in) :: val
    integer, dimension(size(res)) :: tmp
  <Sorting: merge>
  end subroutine merge_int

  subroutine merge_real (res, src1, src2, val)
    integer, dimension(:), intent(out) :: res
    integer, dimension(:), intent(in) :: src1, src2
    real(default), dimension(:), intent(in) :: val
    integer, dimension(size(res)) :: tmp
  <Sorting: merge>
  end subroutine merge_real

<Sorting: merge>≡
  integer :: i1, i2, i
  i1 = 1
  i2 = 1
  do i = 1, size (tmp)
    if (val(src1(i1)) <= val(src2(i2))) then
      tmp(i) = src1(i1); i1 = i1 + 1
      if (i1 > size (src1)) then
        tmp(i+1:) = src2(i2:)
        exit
      end if
    else
      tmp(i) = src2(i2); i2 = i2 + 1
      if (i2 > size (src2)) then
        tmp(i+1:) = src1(i1:)
        exit
      end if
    end if
  end do
  res = tmp

```

## 2.16.2 Concatenating arrays

Not precisely a sorting function, but useful: Concatenate two arrays.

```

<Sorting: public>+≡
  public :: concat

```

```

<Sorting: interfaces>+≡
interface concat
  module procedure concat_int
  module procedure concat_real
end interface

<Sorting: procedures>+≡
function concat_int (val1, val2) result (val12)
  integer, dimension(:), intent(in) :: val1, val2
  integer, dimension(size(val1)+size(val2)) :: val12
<Sorting: concat>
end function concat_int

function concat_real (val1, val2) result (val12)
  real(default), dimension(:), intent(in) :: val1, val2
  integer, dimension(size(val1)+size(val2)) :: val12
<Sorting: concat>
end function concat_real

<Sorting: concat>≡
val12(:size(val1)) = val1
val12(size(val1)+1:) = val2

```

### 2.16.3 Test

```

<Sorting: public>+≡
public :: sorting_test

<Sorting: procedures>+≡
subroutine sorting_test ()
  integer, parameter :: NMAX = 10
  real(default), dimension(NMAX) :: rval
  integer, dimension(NMAX) :: ival
  real, dimension(NMAX) :: harvest
  integer :: i, j
  print *, "Sorting real values:"
  do i = 1, NMAX
    print *
    call random_number (harvest(:i))
    rval(:i) = harvest(:i)
    print "(10(1x,F7.4))", rval(:i)
    rval(:i) = sort (rval(:i))
    print "(10(1x,F7.4))", rval(:i)
    do j = i, 2, -1
      if (rval(j)-rval(j-1) < 0) &
        call msg_fatal ("Sorting failure")
    end do
  end do
  print *
  print *, "Sorting integer values:"
  do i = 1, NMAX
    print *
    call random_number (harvest(:i))

```

```

      ival(:i) = harvest(:i) * NMAX * 2
      print "(10(1x,I2))", ival(:i)
      ival(:i) = sort (ival(:i))
      print "(10(1x,I2))", ival(:i)
      do j = i, 2, -1
        if (ival(j)-ival(j-1) < 0) &
          call msg_fatal ("Sorting failure")
        end do
      end do
end subroutine sorting_test

```

## Chapter 3

# Text handling

WHIZARD has to handle complex structures in input (and output) data. Doing this in a generic and transparent way requires a generic lexer and parser. The necessary modules are implemented here:

**ifiles** Implementation of line-oriented internal files in a more flexible way (linked lists of variable-length strings) than the Fortran builtin features.

**lexers** Read text and transform it into a token stream.

**syntax\_rules** Define the rules for interpreting tokens, to be used by the parser.

**parser** Categorize tokens (keyword, string, number etc.) and use a set of syntax rules to transform the input into a parse tree.

## 3.1 Internal files

The internal files introduced here (`ifile`) are a replacement for the built-in internal files, which are fixed-size arrays of fixed-length character strings. The `ifile` type is a doubly-linked list of variable-length character strings with line numbers.

```
<ifiles.f90>≡  
  <File header>  
  
  module ifiles  
  
    <Use strings>  
    <Use file utils>  
    use limits, only: EOF !NODEP!  
  
    <Standard module head>  
  
    <Ifiles: public>  
  
    <Ifiles: types>  
  
    <Ifiles: interfaces>  
  
    contains  
  
    <Ifiles: subroutines>  
  
  end module ifiles
```

### 3.1.1 iostat codes

```
<Limits: public parameters>+≡  
  integer, parameter, public :: EOF = iostat_end, EOR = iostat_eor
```

### 3.1.2 The line type

The line entry type is for internal use, it is the list entry to be collected in an `ifile` object.

```
<Ifiles: types>≡  
  type :: line_entry_t  
    private  
    type(line_entry_t), pointer :: previous => null ()  
    type(line_entry_t), pointer :: next => null ()  
    type(string_t) :: string  
    integer :: index  
  end type line_entry_t
```

Create a new list entry, given a varying string as input. The line number and pointers are not set, these make sense only within an `ifile`.

```
<Ifiles: subroutines>≡  
  subroutine line_entry_create (line, string)
```



```

        type(line_entry_t), pointer :: line
        type(string_t), intent(in) :: string
        allocate (line)
        line%string = string
    end subroutine line_entry_create

```

Destroy a single list entry: Since the pointer components should not be deallocated explicitly, just deallocate the object itself.

```

<Ifiles: subroutines>+≡
    subroutine line_entry_destroy (line)
        type(line_entry_t), pointer :: line
        deallocate (line)
    end subroutine line_entry_destroy

```

### 3.1.3 The ifile type

The internal file is a linked list of line entries.

```

<Ifiles: public>≡
    public :: ifile_t

<Ifiles: types>+≡
    type :: ifile_t
    private
        type(line_entry_t), pointer :: first => null ()
        type(line_entry_t), pointer :: last => null ()
        integer :: n_lines = 0
    end type ifile_t

```

We need no explicit initializer, but a routine which recursively deallocates the contents may be appropriate. After this, existing line pointers may become undefined, so they should be nullified before the file is destroyed.

```

<Ifiles: public>+≡
    public :: ifile_clear

<Ifiles: subroutines>+≡
    subroutine ifile_clear (ifile)
        type(ifile_t), intent(inout) :: ifile
        type(line_entry_t), pointer :: current
        do while (associated (ifile%first))
            current => ifile%first
            ifile%first => current%next
            call line_entry_destroy (current)
        end do
        nullify (ifile%last)
        ifile%n_lines = 0
    end subroutine ifile_clear

```

The finalizer is just an alias for the above.

```

<Ifiles: public>+≡
    public :: ifile_final

```

```

<Ifiles: interfaces>+=
  interface ifile_final
    module procedure ifile_clear
  end interface

```

### 3.1.4 I/O on ifiles

Fill an ifile from an ordinary external file, i.e., I/O unit. If the ifile is not empty, the old contents will be destroyed. We can read a fixed-length character string, an ISO varying string, an ordinary internal file (character-string array), or from an external unit. In the latter case, lines are appended until EOF is reached. Finally, there is a variant which reads from another ifile, effectively copying it.

```

<Ifiles: public>+=
  public :: ifile_read

<Ifiles: interfaces>+=
  interface ifile_read
    module procedure ifile_read_from_string
    module procedure ifile_read_from_char
    module procedure ifile_read_from_unit
    module procedure ifile_read_from_char_array
    module procedure ifile_read_from_ifile
  end interface

<Ifiles: subroutines>+=
  subroutine ifile_read_from_string (ifile, string)
    type(ifile_t), intent(inout) :: ifile
    type(string_t), intent(in) :: string
    call ifile_clear (ifile)
    call ifile_append (ifile, string)
  end subroutine ifile_read_from_string

  subroutine ifile_read_from_char (ifile, char)
    type(ifile_t), intent(inout) :: ifile
    character(*), intent(in) :: char
    call ifile_clear (ifile)
    call ifile_append (ifile, char)
  end subroutine ifile_read_from_char

  subroutine ifile_read_from_char_array (ifile, char)
    type(ifile_t), intent(inout) :: ifile
    character(*), dimension(:), intent(in) :: char
    call ifile_clear (ifile)
    call ifile_append (ifile, char)
  end subroutine ifile_read_from_char_array

  subroutine ifile_read_from_unit (ifile, unit, iostat)
    type(ifile_t), intent(inout) :: ifile
    integer, intent(in) :: unit
    integer, intent(out), optional :: iostat
    call ifile_clear (ifile)
    call ifile_append (ifile, unit, iostat)
  end subroutine ifile_read_from_unit

```

```

subroutine ifile_read_from_ifile (ifile, ifile_in)
  type(ifile_t), intent(inout) :: ifile
  type(ifile_t), intent(in) :: ifile_in
  call ifile_clear (ifile)
  call ifile_append (ifile, ifile_in)
end subroutine ifile_read_from_ifile

```

Append to an ifile. The same as reading, but without resetting the ifile. In addition, there is a routine for appending a whole ifile.

*(Ifiles: public)*+≡

```

public :: ifile_append

```

*(Ifiles: interfaces)*+≡

```

interface ifile_append
  module procedure ifile_append_from_string
  module procedure ifile_append_from_char
  module procedure ifile_append_from_unit
  module procedure ifile_append_from_char_array
  module procedure ifile_append_from_ifile
end interface

```

*(Ifiles: subroutines)*+≡

```

subroutine ifile_append_from_string (ifile, string)
  type(ifile_t), intent(inout) :: ifile
  type(string_t), intent(in) :: string
  type(line_entry_t), pointer :: current
  call line_entry_create (current, string)
  current%index = ifile%n_lines + 1
  if (associated (ifile%last)) then
    current%previous => ifile%last
    ifile%last%next => current
  else
    ifile%first => current
  end if
  ifile%last => current
  ifile%n_lines = current%index
end subroutine ifile_append_from_string

subroutine ifile_append_from_char (ifile, char)
  type(ifile_t), intent(inout) :: ifile
  character(*), intent(in) :: char
  call ifile_append_from_string (ifile, var_str (trim (char)))
end subroutine ifile_append_from_char

subroutine ifile_append_from_char_array (ifile, char)
  type(ifile_t), intent(inout) :: ifile
  character(*), dimension(:), intent(in) :: char
  integer :: i
  do i = 1, size (char)
    call ifile_append_from_string (ifile, var_str (trim (char(i))))
  end do
end subroutine ifile_append_from_char_array

subroutine ifile_append_from_unit (ifile, unit, iostat)
  type(ifile_t), intent(inout) :: ifile

```

```

integer, intent(in) :: unit
integer, intent(out), optional :: iostat
type(string_t) :: buffer
integer :: ios
ios = 0
READ_LOOP: do
    call get (unit, buffer, iostat = ios)
    if (ios == EOF .or. ios > 0) exit READ_LOOP
    call ifile_append_from_string (ifile, buffer)
end do READ_LOOP
if (present (iostat)) then
    iostat = ios
else if (ios > 0) then
    call get (unit, buffer) ! trigger error again
end if
end subroutine ifile_append_from_unit

subroutine ifile_append_from_ifile (ifile, ifile_in)
type(ifile_t), intent(inout) :: ifile
type(ifile_t), intent(in) :: ifile_in
type(line_entry_t), pointer :: current
current => ifile_in%first
do while (associated (current))
    call ifile_append_from_string (ifile, current%string)
    current => current%next
end do
end subroutine ifile_append_from_ifile

```

Write the ifile contents to an external unit

```

<Ifiles: public>+≡
public :: ifile_write

<Ifiles: subroutines>+≡
subroutine ifile_write (ifile, unit, iostat)
type(ifile_t), intent(in) :: ifile
integer, intent(in), optional :: unit
integer, intent(out), optional :: iostat
integer :: u
type(line_entry_t), pointer :: current
u = output_unit (unit); if (u < 0) return
current => ifile%first
do while (associated (current))
    call put_line (u, current%string, iostat)
    current => current%next
end do
end subroutine ifile_write

```

Convert the ifile to an array of strings, which is allocated by this function:

```

<Ifiles: public>+≡
public :: ifile_to_string_array

<Ifiles: subroutines>+≡
subroutine ifile_to_string_array (ifile, string)
type(ifile_t), intent(in) :: ifile

```

```

type(string_t), dimension(:), intent(inout), allocatable :: string
type(line_entry_t), pointer :: current
integer :: i
allocate (string (ifile_get_length (ifile)))
current => ifile%first
do i = 1, ifile_get_length (ifile)
    string(i) = current%string
    current => current%next
end do
end subroutine ifile_to_string_array

```

### 3.1.5 Ifile tools

```

<Ifiles: public>+≡
public :: ifile_get_length

<Ifiles: subroutines>+≡
function ifile_get_length (ifile) result (length)
    integer :: length
    type(ifile_t), intent(in) :: ifile
    length = ifile%n_lines
end function ifile_get_length

```

### 3.1.6 Line pointers

Instead of the implicit pointer used in ordinary file access, we define explicit pointers, so there can be more than one at a time.

```

<Ifiles: public>+≡
public :: line_p

<Ifiles: types>+≡
type :: line_p
private
type(line_entry_t), pointer :: p => null ()
end type line_p

```

Assign a file pointer to the first or last line in an ifile:

```

<Ifiles: public>+≡
public :: line_init

<Ifiles: subroutines>+≡
subroutine line_init (line, ifile, back)
    type(line_p), intent(inout) :: line
    type(ifile_t), intent(in) :: ifile
    logical, intent(in), optional :: back
    if (present (back)) then
        if (back) then
            line%p => ifile%last
        else
            line%p => ifile%first
        end if
    else

```

```

        line%p => ifile%first
    end if
end subroutine line_init

```

Remove the pointer association:

```

<Ifiles: public>+≡
    public :: line_final
<Ifiles: subroutines>+≡
    subroutine line_final (line)
        type(line_p), intent(inout) :: line
        nullify (line%p)
    end subroutine line_final

```

Go one step forward

```

<Ifiles: public>+≡
    public :: line_advance
<Ifiles: subroutines>+≡
    subroutine line_advance (line)
        type(line_p), intent(inout) :: line
        if (associated (line%p)) line%p => line%p%next
    end subroutine line_advance

```

Go one step backward

```

<Ifiles: public>+≡
    public :: line_backspace
<Ifiles: subroutines>+≡
    subroutine line_backspace (line)
        type(line_p), intent(inout) :: line
        if (associated (line%p)) line%p => line%p%previous
    end subroutine line_backspace

```

Check whether we are accessing a valid line

```

<Ifiles: public>+≡
    public :: line_is_associated
<Ifiles: subroutines>+≡
    function line_is_associated (line) result (ok)
        logical :: ok
        type(line_p), intent(in) :: line
        ok = associated (line%p)
    end function line_is_associated

```

### 3.1.7 Access lines via pointers

We do not need the ifile as an argument to these functions, because the `line` type will point to an existing ifile.

```

<Ifiles: public>+≡
    public :: line_get_string

```

```

<Ifiles: subroutines>+=
function line_get_string (line) result (string)
  type(string_t) :: string
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
    string = line%p%string
  else
    string = ""
  end if
end function line_get_string

```

Variant where the line pointer is advanced after reading.

```

<Ifiles: public>+=
public :: line_get_string_advance
<Ifiles: subroutines>+=
function line_get_string_advance (line) result (string)
  type(string_t) :: string
  type(line_p), intent(inout) :: line
  if (associated (line%p)) then
    string = line%p%string
    call line_advance (line)
  else
    string = ""
  end if
end function line_get_string_advance

```

```

<Ifiles: public>+=
public :: line_get_index
<Ifiles: subroutines>+=
function line_get_index (line) result (index)
  integer :: index
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
    index = line%p%index
  else
    index = 0
  end if
end function line_get_index

```

```

<Ifiles: public>+=
public :: line_get_length
<Ifiles: subroutines>+=
function line_get_length (line) result (length)
  integer :: length
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
    length = len (line%p%string)
  else
    length = 0
  end if
end function line_get_length

```

## 3.2 Lexer

The lexer purpose is to read from a line-separated character input stream (usually a file) and properly chop the stream into lexemes (tokens). [The parser will transform lexemes into meaningful tokens, to be stored in a parse tree, therefore we do not use the term 'token' here.] The input is read line-by-line, but interpreted free-form, except for quotes and the comment syntax. (Fortran 2003 would allow us to use a stream type for reading.)

In an object-oriented approach, we can dynamically create and destroy lexers, including the lexer setup.

The main lexer function is to return a lexeme according to the basic lexer rules (quotes, comments, whitespace, special classes). There is also a routine to write back a lexeme to the input stream (but only once).

For the rules, we separate the possible characters into classes. Whitespace usually consists of blank, tab, and line-feed, where any number of consecutive whitespace is equivalent to one. Quoted strings are enclosed by a pair of quote characters, possibly multiline. Comments are similar to quotes, but interpreted as whitespace. Numbers are identified (not distinguishing real and integer) but not interpreted. Other character classes make up identifiers.

```
<lexers.f90>≡
  <File header>

  module lexers

    <Use strings>
    <Use file utils>
    use limits, only: EOF, EOR !NODEP!
    use limits, only: LF !NODEP!
    use limits, only: WHITESPACE_CHARS, LCLETTERS, UCLETTERS, DIGITS !NODEP!
    use diagnostics !NODEP!
    use ifiles, only: ifile_t
    use ifiles, only: line_p, line_is_associated, line_init, line_final
    use ifiles, only: line_get_string_advance

    <Standard module head>

    <Lexer: public>

    <Lexer: parameters>

    <Lexer: types>

    <Lexer: interfaces>

    contains

    <Lexer: procedures>

  end module lexers
```



### 3.2.1 Input streams

For flexible input, we define a generic stream type that refers to either an external file, an external unit which is already open, a string, an `ifile` object (internal file, i.e., string list), or a line pointer to an `ifile` object. The stream type actually follows the idea of a formatted external file, which is line-oriented. Thus, the stream reader always returns a whole record (input line).

Note that only in the string version, the stream contents are stored inside the stream object. In the `ifile` version, the stream contains only the line pointer, while in the external-file case, the line pointer is implicitly created by the runtime library.

```
<Lexer: public>≡
  public :: stream_t

<Lexer: types>≡
  type :: stream_t
    type(string_t), pointer :: filename => null ()
    integer, pointer :: unit => null ()
    type(string_t), pointer :: string => null ()
    type(ifile_t), pointer :: ifile => null ()
    type(line_p), pointer :: line => null ()
    integer :: record = 0
    logical :: eof = .false.
  end type stream_t
```

The initializers refer to the specific version. The stream should be undefined before calling this.

```
<Lexer: public>+≡
  public :: stream_init

<Lexer: interfaces>≡
  interface stream_init
    module procedure stream_init_filename
    module procedure stream_init_unit
    module procedure stream_init_string
    module procedure stream_init_ifile
    module procedure stream_init_line
  end interface

<Lexer: procedures>≡
  subroutine stream_init_filename (stream, filename)
    type(stream_t), intent(out) :: stream
    character(*), intent(in) :: filename
    integer :: unit
    unit = free_unit ()
    open (unit=unit, file=filename, status="old", action="read")
    call stream_init_unit (stream, unit)
    allocate (stream%filename)
    stream%filename = filename
  end subroutine stream_init_filename

  subroutine stream_init_unit (stream, unit)
    type(stream_t), intent(out) :: stream
    integer, intent(in) :: unit
```

```

        allocate (stream%unit)
        stream%unit = unit
        stream%eof = .false.
    end subroutine stream_init_unit

    subroutine stream_init_string (stream, string)
        type(stream_t), intent(out) :: stream
        type(string_t), intent(in) :: string
        allocate (stream%string)
        stream%string = string
    end subroutine stream_init_string

    subroutine stream_init_ifile (stream, ifile)
        type(stream_t), intent(out) :: stream
        type(ifile_t), intent(in) :: ifile
        type(line_p) :: line
        call line_init (line, ifile)
        call stream_init_line (stream, line)
        allocate (stream%ifile)
        stream%ifile = ifile
    end subroutine stream_init_ifile

    subroutine stream_init_line (stream, line)
        type(stream_t), intent(out) :: stream
        type(line_p), intent(in) :: line
        allocate (stream%line)
        stream%line = line
    end subroutine stream_init_line

```

The finalizer restores the initial state. If an external file was opened, it is closed.

```

<Lexer: public>+≡
    public :: stream_final

<Lexer: procedures>+≡
    subroutine stream_final (stream)
        type(stream_t), intent(inout) :: stream
        if (associated (stream%filename)) then
            close (stream%unit)
            deallocate (stream%unit)
            deallocate (stream%filename)
        else if (associated (stream%unit)) then
            deallocate (stream%unit)
        else if (associated (stream%string)) then
            deallocate (stream%string)
        else if (associated (stream%ifile)) then
            call line_final (stream%line)
            deallocate (stream%line)
            deallocate (stream%ifile)
        else if (associated (stream%line)) then
            call line_final (stream%line)
            deallocate (stream%line)
        end if
    end subroutine stream_final

```

This returns the next record from the input stream. Depending on the stream type, the stream pointers are modified: Reading from external unit, the external file is advanced (implicitly). Reading from string, the string is replaced by an empty string. Reading from ifile, the line pointer is advanced. Note that the *iostat* argument is mandatory.

*<Lexer: public>+≡*

public :: stream\_get\_record

*<Lexer: procedures>+≡*

```
subroutine stream_get_record (stream, string, iostat)
  type(stream_t), intent(inout) :: stream
  type(string_t), intent(out) :: string
  integer, intent(out) :: iostat
  if (associated (stream%unit)) then
    if (stream%eof) then
      iostat = EOF
    else
      call get (stream%unit, string, iostat=iostat)
      if (iostat == EOR) then
        iostat = 0
        stream%record = stream%record + 1
      end if
      if (iostat == EOF) then
        iostat = 0
        stream%eof = .true.
        if (len (string) /= 0) stream%record = stream%record + 1
      end if
    end if
  else if (associated (stream%string)) then
    if (len (stream%string) /= 0) then
      string = stream%string
      stream%string = ""
      iostat = 0
      stream%record = stream%record + 1
    else
      string = ""
      iostat = EOF
    end if
  else if (associated (stream%line)) then
    if (line_is_associated (stream%line)) then
      string = line_get_string_advance (stream%line)
      iostat = 0
      stream%record = stream%record + 1
    else
      string = ""
      iostat = EOF
    end if
  else
    call msg_bug (" Attempt to read from uninitialized input stream")
  end if
end subroutine stream_get_record
```

Return the current stream source as a message string.

*<Lexer: public>+≡*

```

public :: stream_get_source_info_string
<Lexer: procedures>+≡
function stream_get_source_info_string (stream) result (string)
  type(string_t) :: string
  type(stream_t), intent(in) :: stream
  character(20) :: buffer
  if (associated (stream%filename)) then
    string = "File '" // stream%filename // "' (unit = "
    write (buffer, "(I0)") stream%unit
    string = string // trim (buffer) // ")"
  else if (associated (stream%unit)) then
    write (buffer, "(I0)") stream%unit
    string = "Unit " // trim (buffer)
  else if (associated (stream%string)) then
    string = "Input string"
  else if (associated (stream%ifile) .or. associated (stream%line)) then
    string = "Internal file"
  else
    string = ""
  end if
end function stream_get_source_info_string

```

Return the index of the record just read as a message string.

```

<Lexer: public>+≡
public :: stream_get_record_info_string
<Lexer: procedures>+≡
function stream_get_record_info_string (stream) result (string)
  type(string_t) :: string
  type(stream_t), intent(in) :: stream
  character(20) :: buffer
  string = stream_get_source_info_string (stream)
  if (string /= "") string = string // ", "
  write (buffer, "(I0)") stream%record
  string = string // "line " // trim (buffer)
end function stream_get_record_info_string

```

### 3.2.2 Keyword list

The lexer should be capable of identifying a token as a known keyword. To this end, we store a list of keywords:

```

<Lexer: public>+≡
public :: keyword_list_t
<Lexer: types>+≡
type :: keyword_entry_t
  private
  type(string_t) :: string
  type(keyword_entry_t), pointer :: next => null ()
end type keyword_entry_t

type :: keyword_list_t

```

```

private
type(keyword_entry_t), pointer :: first => null ()
type(keyword_entry_t), pointer :: last => null ()
end type keyword_list_t

```

Add a new string to the keyword list, unless it is already there:

```

<Lexer: public>+≡
public :: keyword_list_add

<Lexer: procedures>+≡
subroutine keyword_list_add (keylist, string)
type(keyword_list_t), intent(inout) :: keylist
type(string_t), intent(in) :: string
type(keyword_entry_t), pointer :: k_entry_new
if (.not. keyword_list_contains (keylist, string)) then
allocate (k_entry_new)
k_entry_new%string = string
if (associated (keylist%first)) then
keylist%last%next => k_entry_new
else
keylist%first => k_entry_new
end if
keylist%last => k_entry_new
end if
end subroutine keyword_list_add

```

Return true if a string is a keyword.

```

<Lexer: public>+≡
public :: keyword_list_contains

<Lexer: procedures>+≡
function keyword_list_contains (keylist, string) result (found)
type(keyword_list_t), intent(in) :: keylist
type(string_t), intent(in) :: string
logical :: found
found = .false.
call check_rec (keylist%first)
contains
recursive subroutine check_rec (k_entry)
type(keyword_entry_t), pointer :: k_entry
if (associated (k_entry)) then
if (k_entry%string /= string) then
call check_rec (k_entry%next)
else
found = .true.
end if
end if
end subroutine check_rec
end function keyword_list_contains

```

Write the keyword list

```

<Lexer: public>+≡
public :: keyword_list_write

```

```

<Lexer: interfaces>+≡
    interface keyword_list_write
        module procedure keyword_list_write_unit
    end interface

<Lexer: procedures>+≡
    subroutine keyword_list_write_unit (keylist, unit)
        type(keyword_list_t), intent(in) :: keylist
        integer, intent(in) :: unit
        write (unit, "(A)") "Keyword list:"
        if (associated (keylist%first)) then
            call keyword_write_rec (keylist%first)
            write (unit, *)
        else
            write (unit, "(1x,A)") "[empty]"
        end if
    contains
        recursive subroutine keyword_write_rec (k_entry)
            type(keyword_entry_t), intent(in), pointer :: k_entry
            if (associated (k_entry)) then
                write (unit, "(1x,A)", advance="no") char (k_entry%string)
                call keyword_write_rec (k_entry%next)
            end if
        end subroutine keyword_write_rec
    end subroutine keyword_list_write_unit

```

Clear the keyword list

```

<Lexer: public>+≡
    public :: keyword_list_final

<Lexer: procedures>+≡
    subroutine keyword_list_final (keylist)
        type(keyword_list_t), intent(inout) :: keylist
        call keyword_destroy_rec (keylist%first)
        nullify (keylist%last)
    contains
        recursive subroutine keyword_destroy_rec (k_entry)
            type(keyword_entry_t), pointer :: k_entry
            if (associated (k_entry)) then
                call keyword_destroy_rec (k_entry%next)
                deallocate (k_entry)
            end if
        end subroutine keyword_destroy_rec
    end subroutine keyword_list_final

```

### 3.2.3 Lexeme templates

This type is handled like a rudimentary regular expression. It determines the lexer behavior when matching a string. The actual objects made from this type and the corresponding matching routines are listed below.

```

<Lexer: types>+≡
    type :: template_t

```

```

private
integer :: type
character(256) :: charset1, charset2
integer :: len1, len2
end type template_t

```

These are the types that valid lexemes can have:

```

<Lexer: public>+≡
public :: T_KEYWORD, T_IDENTIFIER, T_QUOTED, T_NUMERIC

<Lexer: parameters>≡
integer, parameter :: T_KEYWORD = 1
integer, parameter :: T_IDENTIFIER = 2, T_QUOTED = 3, T_NUMERIC = 4

```

These are special types:

```

<Lexer: parameters>+≡
integer, parameter :: EMPTY = 0, WHITESPACE = 10
integer, parameter :: NO_MATCH = 11, IO_ERROR = 12, OVERFLOW = 13
integer, parameter :: UNMATCHED_QUOTE = 14

```

In addition, we have EOF which is a negative integer, normally  $-1$ . Printout for debugging:

```

<Lexer: procedures>+≡
subroutine lexeme_type_write (type, unit)
integer, intent(in) :: type
integer, intent(in) :: unit
select case (type)
case (EMPTY);      write(unit,"(A)",advance="no") " EMPTY      "
case (WHITESPACE); write(unit,"(A)",advance="no") " WHITESPACE "
case (T_IDENTIFIER);write(unit,"(A)",advance="no") " IDENTIFIER "
case (T_QUOTED);   write(unit,"(A)",advance="no") " QUOTED      "
case (T_NUMERIC);  write(unit,"(A)",advance="no") " NUMERIC      "
case (IO_ERROR);   write(unit,"(A)",advance="no") " IO_ERROR    "
case (OVERFLOW);   write(unit,"(A)",advance="no") " OVERFLOW    "
case (UNMATCHED_QUOTE); write(unit,"(A)",advance="no") " UNMATCHEDQ "
case (NO_MATCH);   write(unit,"(A)",advance="no") " NO_MATCH    "
case (EOF);        write(unit,"(A)",advance="no") " EOF          "
case default;      write(unit,"(A)",advance="no") " [illegal]   "
end select
end subroutine lexeme_type_write

subroutine template_write (tt, unit)
type(template_t), intent(in) :: tt
integer, intent(in) :: unit
call lexeme_type_write (tt%type, unit)
write (unit, "(A)", advance="no") " '" // tt%charset1(1:tt%len1) // "'"
write (unit, "(A)", advance="no") " '" // tt%charset2(1:tt%len2) // "'"
end subroutine template_write

```

The matching functions all return the number of matched characters in the provided string. If this number is zero, the match has failed.

The `template` functions are declared `pure` because they appear in `forall` loops below.

A template for whitespace:

```

<Lexer: procedures>+≡
  pure function template_whitespace (chars) result (tt)
    character(*), intent(in) :: chars
    type(template_t) :: tt
    tt = template_t (WHITESPACE, chars, "", len (chars), 0)
  end function template_whitespace

```

Just match the string against the character set.

```

<Lexer: procedures>+≡
  subroutine match_whitespace (tt, s, n)
    type(template_t), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    n = verify (s, tt%charset1(1:tt%len1)) - 1
    if (n < 0) n = len (s)
  end subroutine match_whitespace

```

A template for normal identifiers. To match, a lexeme should have a first character in class `chars1` and an arbitrary number of further characters in class `chars2`. If the latter is empty, we are looking for a single-character lexeme.

```

<Lexer: procedures>+≡
  pure function template_identifier (chars1, chars2) result (tt)
    character(*), intent(in) :: chars1, chars2
    type(template_t) :: tt
    tt = template_t (T_IDENTIFIER, chars1, chars2, len(chars1), len(chars2))
  end function template_identifier

```

Here, the first letter must match, the others may or may not.

```

<Lexer: procedures>+≡
  subroutine match_identifier (tt, s, n)
    type(template_t), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    if (verify (s(1:1), tt%charset1(1:tt%len1)) == 0) then
      n = verify (s(2:), tt%charset2(1:tt%len2))
      if (n == 0) n = len (s)
    else
      n = 0
    end if
  end subroutine match_identifier

```

A template for quoted strings. The same template applies for comments. The first character set indicates the left quote (could be a sequence of several characters), the second one the matching right quote.

```

<Lexer: procedures>+≡
  pure function template_quoted (chars1, chars2) result (tt)
    character(*), intent(in) :: chars1, chars2
    type(template_t) :: tt
    tt = template_t (T_QUOTED, chars1, chars2, len (chars1), len (chars2))
  end function template_quoted

```



Here, the beginning of the string must exactly match the first character set, then we look for the second one. If found, return. If there is a first quote but no second one, return a negative number, indicating this error condition.

```

<Lexer: procedures>+≡
subroutine match_quoted (tt, s, n, range)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  character(tt%len1) :: ch1
  character(tt%len2) :: ch2
  integer :: i
  ch1 = tt%charset1
  if (s(1:tt%len1) == ch1) then
    ch2 = tt%charset2
    do i = tt%len1 + 1, len (s) - tt%len2 + 1
      if (s(i:i+tt%len2-1) == ch2) then
        n = i + tt%len2 - 1
        range(1) = tt%len1 + 1
        range(2) = i - 1
        return
      end if
    end do
    n = -1
    range = 0
  else
    n = 0
    range = 0
  end if
end subroutine match_quoted

```

A template for real numbers. The first character set is the set of allowed exponent letters. In accordance with the other functions we return the lexeme as a string but do not read it.

```

<Lexer: procedures>+≡
pure function template_numeric (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (T_NUMERIC, chars, "", len (chars), 0)
end function template_numeric

```

A numeric lexeme may be real or integer. We purposely do not allow for a preceding sign. If the number is followed by an exponent, this is included, otherwise the rest is ignored.

There is a possible pitfall with this behavior: while the string `1e3` will be interpreted as a single number, the analogous string `1a3` will be split into the number `1` and an identifier `a3`. There is no easy way around such an ambiguity. We should make sure that the syntax does not contain identifiers like `a3` or `e3`.

```

<Lexer: procedures>+≡
subroutine match_numeric (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s

```

```

integer, intent(out) :: n
integer :: i, n0
character(10), parameter :: digits = "0123456789"
character(2), parameter :: signs = "-+"
n = verify (s, digits) - 1
if (n < 0) then
    n = 0
    return
else if (s(n+1:n+1) == ".") then
    i = verify (s(n+2:), digits) - 1
    if (i < 0) then
        n = len (s)
        return
    else if (i > 0 .or. n > 0) then
        n = n + 1 + i
    end if
end if
n0 = n
if (n > 0) then
    if (verify (s(n+1:n+1), tt%charset1(1:tt%len1)) == 0) then
        n = n + 1
        if (verify (s(n+1:n+1), signs) == 0) n = n + 1
        i = verify (s(n+1:), digits) - 1
        if (i < 0) then
            n = len (s)
        else if (i == 0) then
            n = n0
        else
            n = n + i
        end if
    end if
end if
end subroutine match_numeric

```

The generic matching routine. With Fortran 2003 we would define separate types and use a SELECT TYPE instead.

*<Lexer: procedures>+≡*

```

subroutine match_template (tt, s, n, range)
    type(template_t), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    integer, dimension(2), intent(out) :: range
    select case (tt%type)
    case (WHITESPACE)
        call match_whitespace (tt, s, n)
        range = 0
    case (T_IDENTIFIER)
        call match_identifier (tt, s, n)
        range(1) = 1
        range(2) = len_trim (s)
    case (T_QUOTED)
        call match_quoted (tt, s, n, range)
    case (T_NUMERIC)

```

```

        call match_numeric (tt, s, n)
        range(1) = 1
        range(2) = len_trim (s)
    case default
        call msg_bug ("Invalid lexeme template encountered")
    end select
end subroutine match_template

```

Match against an array of templates. Return the index of the first template that matches together with the number of characters matched and the range of the relevant substring. If all fails, these numbers are zero.

*<Lexer: procedures>+≡*

```

subroutine match (tt, s, n, range, ii)
    type(template_t), dimension(:), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    integer, dimension(2), intent(out) :: range
    integer, intent(out) :: ii
    integer :: i
    do i = 1, size (tt)
        call match_template (tt(i), s, n, range)
        if (n /= 0) then
            ii = i
            return
        end if
    end do
    n = 0
    ii = 0
end subroutine match

```

### 3.2.4 The lexer setup

This object contains information about character classes. As said above, one class consists of quoting chars (matching left and right), another one of comment chars (similar), a class of whitespace, and several classes of characters that make up identifiers. When creating the lexer setup, the character classes are transformed into lexeme templates which are to be matched in a certain predefined order against the input stream.

BLANK should always be taken as whitespace, some things may depend on this. TAB is also fixed by convention, but may in principle be modified. Newline (DOS!) and linefeed are also defined as whitespace.

*<Limits: public parameters>+≡*

```

character, parameter, public :: BLANK = ' ', TAB = achar(9)
character, parameter, public :: CR = achar(13), LF = achar(10)
character, parameter, public :: BACKSLASH = achar(92)
character(*), parameter, public :: WHITESPACE_CHARS = BLANK// TAB // CR // LF
character(*), parameter, public :: LCLETTERS = "abcdefghijklmnopqrstuvwxyz"
character(*), parameter, public :: UCLETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
character(*), parameter, public :: DIGITS = "0123456789"

```

The lexer setup, containing the list of lexeme templates. No defaults yet. The type with index zero will be assigned to the NO\_MATCH lexeme.

The keyword list is not stored, just a pointer to it. We anticipate that the keyword list is part of the syntax table, and the lexer needs not alter it. Furthermore, the lexer is typically finished before the syntax table is.

```

<Lexer: parameters>+≡
    integer, parameter :: CASE_KEEP = 0, CASE_UP = 1, CASE_DOWN = 2

<Lexer: types>+≡
    type :: lexer_setup_t
    private
    type(template_t), dimension(:), allocatable :: tt
    integer, dimension(:), allocatable :: type
    integer :: keyword_case = CASE_KEEP
    type(keyword_list_t), pointer :: keyword_list => null ()
end type lexer_setup_t

```

Fill the lexer setup object. Some things are hardcoded here (whitespace, alphanumeric identifiers), some are free: comment chars (but these must be single, and comments must be terminated by line-feed), quote chars and matches (must be single), characters to be read as one-character lexeme, special classes (characters of one class that should be glued together as identifiers).

```

<Lexer: procedures>+≡
    subroutine lexer_setup_init (setup, &
        comment_chars, quote_chars, quote_match, &
        single_chars, special_class, &
        keyword_list, upper_case_keywords)
    type(lexer_setup_t), intent(inout) :: setup
    character(*), intent(in) :: comment_chars
    character(*), intent(in) :: quote_chars, quote_match
    character(*), intent(in) :: single_chars
    character(*), dimension(:), intent(in) :: special_class
    type(keyword_list_t), pointer :: keyword_list
    logical, intent(in), optional :: upper_case_keywords
    integer :: n, i
    if (present (upper_case_keywords)) then
        if (upper_case_keywords) then
            setup%keyword_case = CASE_UP
        else
            setup%keyword_case = CASE_DOWN
        end if
    else
        setup%keyword_case = CASE_KEEP
    end if
    n = 1 + len (comment_chars) + len (quote_chars) + 1 &
        + len (single_chars) + size (special_class) + 1
    allocate (setup%tt(n))
    allocate (setup%type(0:n))
    n = 0
    setup%type(n) = NO_MATCH
    n = n + 1
    setup%tt(n) = template_whitespace (WHITESPACE_CHARS)

```

```

setup%type(n) = EMPTY
forall (i = 1:len(comment_chars))
    setup%tt(n+i) = template_quoted (comment_chars(i:i), LF)
    setup%type(n+i) = EMPTY
end forall
n = n + len (comment_chars)
forall (i = 1:len(quote_chars))
    setup%tt(n+i) = template_quoted (quote_chars(i:i), quote_match(i:i))
    setup%type(n+i) = T_QUOTED
end forall
n = n + len (quote_chars)
setup%tt(n+1) = template_numeric ("EeDd")
setup%type(n+1) = T_NUMERIC
n = n + 1
forall (i = 1:len (single_chars))
    setup%tt(n+i) = template_identifier (single_chars(i:i), "")
    setup%type(n+i) = T_IDENTIFIER
end forall
n = n + len (single_chars)
forall (i = 1:size (special_class))
    setup%tt(n+i) = template_identifier &
        (trim (special_class(i)), trim (special_class(i)))
    setup%type(n+i) = T_IDENTIFIER
end forall
n = n + size (special_class)
setup%tt(n+1) = template_identifier &
    (LCLETTERS//UCLETTERS, LCLETTERS//DIGITS//"_"/UCLETTERS)
setup%type(n+1) = T_IDENTIFIER
n = n + 1
if (n /= size (setup%tt)) &
    call msg_bug ("Size mismatch in lexer setup")
setup%keyword_list => keyword_list
end subroutine lexer_setup_init

```

The destructor is needed only if the object is not itself part of an allocatable array

```

<Lexer: procedures>+≡
subroutine lexer_setup_final (setup)
    type(lexer_setup_t), intent(inout) :: setup
    deallocate (setup%tt, setup%type)
    setup%keyword_list => null ()
end subroutine lexer_setup_final

```

For debugging: Write the lexer setup

```

<Lexer: procedures>+≡
subroutine lexer_setup_write (setup, unit)
    type(lexer_setup_t), intent(in) :: setup
    integer, intent(in) :: unit
    integer :: i
    write (unit, "(A)") "Lexer setup:"
    if (allocated (setup%tt)) then
        do i = 1, size (setup%tt)
            call template_write (setup%tt(i), unit)
        end do
    end if
end subroutine lexer_setup_write

```

```

        write (unit, '(A)', advance = "no") " -> "
        call lexeme_type_write (setup%type(i), unit)
        write (unit, *)
    end do
else
    write (unit, *) "[empty]"
end if
if (associated (setup%keyword_list)) then
    call keyword_list_write (setup%keyword_list, unit)
end if
end subroutine lexer_setup_write

```

### 3.2.5 The lexeme type

An object of this type is returned by the lexer. Apart from the lexeme string, it gives information about the relevant substring (first and last character index) and the lexeme type. Interpreting the string is up to the parser.

```

<Lexer: public>+≡
    public :: lexeme_t

<Lexer: types>+≡
    type :: lexeme_t
    private
        integer :: type = EMPTY
        type(string_t) :: s
        integer :: b = 0, e = 0
    end type lexeme_t

```

Debugging aid:

```

<Lexer: public>+≡
    public :: lexeme_write

<Lexer: procedures>+≡
    subroutine lexeme_write (t, unit)
        type(lexeme_t), intent(in) :: t
        integer, intent(in) :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        select case (t%type)
        case (T_KEYWORD)
            write (u, *) "KEYWORD:  ' " // char (t%s) // "' "
        case (T_IDENTIFIER)
            write (u, *) "IDENTIFIER: ' " // char (t%s) // "' "
        case (T_QUOTED)
            write (u, *) "QUOTED:    ' " // char (t%s) // "' "
        case (T_NUMERIC)
            write (u, *) "NUMERIC:   ' " // char (t%s) // "' "
        case (UNMATCHED_QUOTE)
            write (u, *) "Unmatched quote: "// char (t%s)
        case (OVERFLOW); write (u, *) "Overflow: "// char (t%s)
        case (EMPTY);   write (u, *) "Empty lexeme"
        case (NO_MATCH); write (u, *) "No match"
        case (IO_ERROR); write (u, *) "IO error"
    end subroutine lexeme_write

```

```

        case (EOF);      write (u, *) "EOF"
        case default
            write (u, *) "Error"
        end select
    end subroutine lexeme_write

```

Store string and type in a lexeme. The range determines the beginning and end of the relevant part of the string. Check for a keyword.

*<Lexer: procedures>+≡*

```

subroutine lexeme_set (t, keyword_list, s, range, type, keyword_case)
    type.lexeme_t, intent(out) :: t
    type(keyword_list_t), pointer :: keyword_list
    type(string_t), intent(in) :: s
    type(string_t) :: keyword
    integer, dimension(2), intent(in) :: range
    integer, intent(in) :: type
    integer, intent(in), optional :: keyword_case
    t%type = type
    if (present (keyword_case)) then
        select case (keyword_case)
            case (CASE_KEEP);   keyword = s
            case (CASE_UP);     keyword = upper_case (s)
            case (CASE_DOWN);   keyword = lower_case (s)
        end select
    else
        keyword = s
    end if
    if (type == T_IDENTIFIER) then
        if (associated (keyword_list)) then
            if (keyword_list_contains (keyword_list, keyword)) &
                t%type = T_KEYWORD
            end if
        end if
        select case (t%type)
            case (T_KEYWORD);   t%s = keyword
            case default;       t%s = s
        end select
        t%b = range(1)
        t%e = range(2)
    end subroutine lexeme_set

subroutine lexeme_clear (t)
    type.lexeme_t, intent(out) :: t
    t%type = EMPTY
    t%s = ""
end subroutine lexeme_clear

```

Retrieve the lexeme string, the relevant part of it, and the type. The last function returns true if there is a break condition reached (error or EOF).

*<Lexer: public>+≡*

```

public :: lexeme_get_string
public :: lexeme_get_contents
public :: lexeme_get_delimiters

```

```

public :: lexeme_get_type
<Lexer: procedures>+≡
function lexeme_get_string (t) result (s)
  type(string_t) :: s
  type(lexeme_t), intent(in) :: t
  s = t%s
end function lexeme_get_string

function lexeme_get_contents (t) result (s)
  type(string_t) :: s
  type(lexeme_t), intent(in) :: t
  s = extract (t%s, t%b, t%e)
end function lexeme_get_contents

function lexeme_get_delimiters (t) result (del)
  type(string_t), dimension(2) :: del
  type(lexeme_t), intent(in) :: t
  del(1) = extract (t%s, finish = t%b-1)
  del(2) = extract (t%s, start = t%e+1)
end function lexeme_get_delimiters

function lexeme_get_type (t) result (type)
  integer :: type
  type(lexeme_t), intent(in) :: t
  type = t%type
end function lexeme_get_type

```

Check for a generic break condition (error/eof) and for eof in particular.

```

<Lexer: public>+≡
public :: lexeme_is_break
public :: lexeme_is_eof

<Lexer: procedures>+≡
function lexeme_is_break (t) result (break)
  logical :: break
  type(lexeme_t), intent(in) :: t
  select case (t%type)
    case (EOF, IO_ERROR, OVERFLOW, NO_MATCH)
      break = .true.
    case default
      break = .false.
  end select
end function lexeme_is_break

function lexeme_is_eof (t) result (ok)
  logical :: ok
  type(lexeme_t), intent(in) :: t
  ok = t%type == EOF
end function lexeme_is_eof

```



### 3.2.6 The lexer object

We store the current lexeme and the current line. The line buffer is set each time a new line is read from file. The working buffer has one character more, to hold any trailing blank. Pointers to line and column are for debugging, they will be used to make up readable error messages for the parser.

*<Lexer: public>+≡*

public :: lexer\_t

*<Lexer: types>+≡*

type :: lexer\_t

private

type(lexer\_setup\_t) :: setup

type(stream\_t), pointer :: stream => null ()

type(lexeme\_t) :: lexeme

type(string\_t) :: previous\_line2

type(string\_t) :: previous\_line1

type(string\_t) :: current\_line

integer :: lines\_read = 0

integer :: current\_column = 0

integer :: previous\_column = 0

type(string\_t) :: buffer

type(lexer\_t), pointer :: parent => null ()

end type lexer\_t

Create-setup wrapper

*<Lexer: public>+≡*

public :: lexer\_init

*<Lexer: procedures>+≡*

subroutine lexer\_init (lexer, &

comment\_chars, quote\_chars, quote\_match, &

single\_chars, special\_class, &

keyword\_list, upper\_case\_keywords, &

parent)

type(lexer\_t), intent(inout) :: lexer

character(\*), intent(in) :: comment\_chars

character(\*), intent(in) :: quote\_chars, quote\_match

character(\*), intent(in) :: single\_chars

character(\*), dimension(:), intent(in) :: special\_class

type(keyword\_list\_t), pointer :: keyword\_list

logical, intent(in), optional :: upper\_case\_keywords

type(lexer\_t), target, intent(in), optional :: parent

call lexer\_setup\_init (lexer%setup, &

comment\_chars = comment\_chars, &

quote\_chars = quote\_chars, &

quote\_match = quote\_match, &

single\_chars = single\_chars, &

special\_class = special\_class, &

keyword\_list = keyword\_list, &

upper\_case\_keywords = upper\_case\_keywords)

if (present (parent)) lexer%parent => parent

call lexer\_clear (lexer)

end subroutine lexer\_init

Clear the lexer state, but not the setup. This should be done when the lexing starts, but it is not known whether the lexer was used before.

```

<Lexer: public>+≡
    public :: lexer_clear

<Lexer: procedures>+≡
    subroutine lexer_clear (lexer)
        type(lexer_t), intent(inout) :: lexer
        call lexeme_clear (lexer%lexeme)
        lexer%previous_line2 = ""
        lexer%previous_line1 = ""
        lexer%current_line = ""
        lexer%lines_read = 0
        lexer%current_column = 0
        lexer%previous_column = 0
        lexer%buffer = ""
    end subroutine lexer_clear

```

Reset lexer state and delete setup

```

<Lexer: public>+≡
    public :: lexer_final

<Lexer: procedures>+≡
    subroutine lexer_final (lexer)
        type(lexer_t), intent(inout) :: lexer
        call lexer_clear (lexer)
        call lexer_setup_final (lexer%setup)
    end subroutine lexer_final

```

### 3.2.7 The lexer routine

For lexing we need to associate an input stream to the lexer.

```

<Lexer: public>+≡
    public :: lexer_assign_stream

<Lexer: procedures>+≡
    subroutine lexer_assign_stream (lexer, stream)
        type(lexer_t), intent(inout) :: lexer
        type(stream_t), intent(in), target :: stream
        lexer%stream => stream
    end subroutine lexer_assign_stream

```

The lexer. The lexer function takes the lexer and returns the currently stored lexeme. If there is none, it is read from buffer, matching against the lexeme templates in the lexer setup. Empty lexemes, i.e., comments and whitespace, are discarded and the buffer is read again until we have found a nonempty lexeme (which may also be EOF or an error condition).

The initial state of the lexer contains an empty lexeme, so reading from buffer is forced. The empty state is restored after returning the lexeme. A nonempty lexeme is present in the lexer only if `lex_back` has been executed before.

The workspace is the `lexer%buffer`, treated as a sort of input stream. We chop off lexemes from the beginning, adjusting the buffer to the left. Whenever the buffer is empty, or we are matching against an open quote which has not terminated, we read a new line and append it to the right. This may result in special conditions, which for simplicity are also returned as lexemes: I/O error, buffer overflow, end of file. If the latter happens during reading a quoted string, we return an unmatched-quote lexeme. Obviously, the special-condition lexemes have to be caught by the parser.

Note that reading further lines is only necessary when reading a quoted string. Otherwise, the line-feed that ends each line is interpreted as whitespace which terminates a preceding lexeme, so there are no other valid multiline lexemes.

To enable meaningful error messages, we also keep track of the line number of the last line read, and the beginning and the end of the current lexeme with respect to this line.

The lexer is implemented as a function that returns the next lexeme (i.e., token). It uses the `lexer` setup and modifies the buffers and pointers stored within the lexer, a side effect. The lexer reads from an input stream object, which also is modified by this reading, e.g., a line pointer is advanced.

```

(Lexer: public)+≡
    public :: lex

(Lexer: procedures)+≡
    subroutine lex (lexeme, lexer)
        type (lexeme_t), intent(out) :: lexeme
        type (lexer_t), intent(inout) :: lexer
        integer :: iostat1, iostat2
        integer :: pos
        integer, dimension(2) :: range
        integer :: template_index, type
        if (.not. associated (lexer%stream)) &
            call msg_bug ("Lexer called without assigned stream")
        GET_LEXEME: do while (lexeme_get_type (lexer%lexeme) == EMPTY)
            if (len (lexer%buffer) /= 0) then
                iostat1 = 0
            else
                call lexer_read_line (lexer, iostat1)
            end if
            select case (iostat1)
            case (0)
                MATCH_BUFFER: do
                    call match (lexer%setup%tt, char (lexer%buffer), &
                                pos, range, template_index)
                    if (pos >= 0) then
                        type = lexer%setup%type(template_index)
                        exit MATCH_BUFFER
                    else
                        pos = 0
                        call lexer_read_line (lexer, iostat2)
                        select case (iostat2)
                        case (EOF); type = UNMATCHED_QUOTE; exit MATCH_BUFFER
                        case (1);  type = IO_ERROR;      exit MATCH_BUFFER
                        case (2);  type = OVERFLOW;       exit MATCH_BUFFER

```

```

        end select
    end if
    end do MATCH_BUFFER
    case (EOF); type = EOF
    case (1);   type = IO_ERROR
    case (2);   type = OVERFLOW
    end select
    call lexeme_set (lexer%lexeme, lexer%setup%keyword_list, &
        extract (lexer%buffer, finish=pos), range, type, &
        lexer%setup%keyword_case)
    lexer%buffer = remove (lexer%buffer, finish=pos)
    lexer%previous_column = lexer%current_column
    lexer%current_column = lexer%current_column + pos
end do GET_LEXEME
lexeme = lexer%lexeme
call lexeme_clear (lexer%lexeme)
end subroutine lex

```

Read a line and append it to the input buffer. If the input buffer overflows, return `iostat=2`. Otherwise, `iostat=1` indicates an I/O error, and `iostat=-1` the EOF.

The input stream may either be an external unit or a `ifile` object. In the latter case, a line is read and the line pointer is advanced.

Note that inserting LF between input lines is the Unix convention. Since we are doing this explicitly when gluing lines together, we can pattern-match against LF without having to worry about the system.

```

<Lexer: procedures>+≡
subroutine lexer_read_line (lexer, iostat)
    type(lexer_t), intent(inout) :: lexer
    integer, intent(out) :: iostat
    type(string_t) :: current_line
    current_line = lexer%current_line
    call stream_get_record (lexer%stream, lexer%current_line, iostat)
    if (iostat == 0) then
        lexer%lines_read = lexer%lines_read + 1
        lexer%previous_line2 = lexer%previous_line1
        lexer%previous_line1 = current_line
        lexer%buffer = lexer%buffer // lexer%current_line // LF
        lexer%previous_column = 0
        lexer%current_column = 0
    end if
end subroutine lexer_read_line

```

Once in a while we have read one lexeme to many, which can be pushed back into the input stream. Do not do this more than once.

```

<Lexer: public>+≡
public :: lexer_put_back

<Lexer: procedures>+≡
subroutine lexer_put_back (lexer, lexeme)
    type(lexer_t), intent(inout) :: lexer
    type(lexeme_t), intent(in) :: lexeme
    if (lexeme_get_type (lexer%lexeme) == EMPTY) then

```

```

        lexer%lexeme = lexeme
    else
        call msg_bug (" Lexer: lex_back fails; probably called twice")
    end if
end subroutine lexer_put_back

```

### 3.2.8 Diagnostics

For debugging: print just the setup

```

<Lexer: public>+≡
    public :: lexer_write_setup

<Lexer: procedures>+≡
    subroutine lexer_write_setup (lexer, unit)
        type(lexer_t), intent(in) :: lexer
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        call lexer_setup_write (lexer%setup, u)
    end subroutine lexer_write_setup

```

This is useful for error printing: show the current line with index and a pointer to the current column within the line.

```

<Lexer: public>+≡
    public :: lexer_show_location

<Lexer: procedures>+≡
    subroutine lexer_show_location (lexer)
        type(lexer_t), intent(in) :: lexer
        type(string_t) :: loc_str
        if (associated (lexer%parent)) then
            call lexer_show_source (lexer%parent)
            call msg_message ("[includes]")
        else
            call msg_message ()
        end if
        if (associated (lexer%stream)) then
            call msg_message &
                (char (stream_get_record_info_string (lexer%stream)) // ":")
        end if
        if (lexer%lines_read >= 4) call msg_result ("[...]")
        if (lexer%lines_read >= 3) call msg_result (char (lexer%previous_line2))
        if (lexer%lines_read >= 2) call msg_result (char (lexer%previous_line1))
        if (lexer%lines_read >= 1) then
            call msg_result (char (lexer%current_line))
            loc_str = repeat (" ", lexer%previous_column)
            loc_str = loc_str // "^"
            if (lexer%current_column > lexer%previous_column) then
                loc_str = loc_str &
                    // repeat ("-", max (lexer%current_column &
                        - lexer%previous_column - 1, 0)) &
                    // "^"
            end if
        end if
    end subroutine lexer_show_location

```

```

        call msg_result (char (loc_str))
    end if
end subroutine lexer_show_location

```

This just prints the current stream source.

*<Lexer: procedures>+≡*

```

recursive subroutine lexer_show_source (lexer)
    type(lexer_t), intent(in) :: lexer
    type(string_t) :: loc_str
    if (associated (lexer%parent)) then
        call lexer_show_source (lexer%parent)
        call msg_message ("[includes]")
    else
        call msg_message ()
    end if
    if (associated (lexer%stream)) then
        call msg_message &
            (char (stream_get_source_info_string (lexer%stream)) // ":")
    end if
end subroutine lexer_show_source

```

Test the lexer by lexing and printing all lexemes from unit u, one per line, using preset conventions

*<Lexer: public>+≡*

```

public :: lexer_test

```

*<Lexer: procedures>+≡*

```

subroutine lexer_test (lexer, unit)
    type(lexer_t), intent(inout) :: lexer
    integer, intent(in) :: unit
    type(stream_t), target :: stream
    type(string_t) :: string
    type(lexeme_t) :: lexeme
    string = "abcdefghij"
    call lexer_init (lexer, &
        comment_chars = "", &
        quote_chars = "<'\"", &
        quote_match = ">'\"", &
        single_chars = "?*+=,()", &
        special_class = ("/ "." /), &
        keyword_list = null ())
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    do
        call lex (lexeme, lexer)
        call lexeme_write (lexeme, unit)
        if (lexeme_is_break (lexeme)) exit
    end do
    call stream_final (stream)
    call lexer_final (lexer)
end subroutine lexer_test

```

### 3.3 Syntax rules

This module provides tools to handle syntax rules in an abstract way.

```
<syntax_rules.f90>≡  
<File header>  
  
module syntax_rules  
  
  <Use strings>  
  <Use file utils>  
  use limits, only: UNQUOTED !NODEP!  
  use diagnostics !NODEP!  
  use ifiles, only: line_p, line_init, line_get_string_advance, line_final  
  use ifiles, only: ifile_t, ifile_get_length  
  use lexers  
  
  <Standard module head>  
  
  <Syntax: public>  
  
  <Syntax: parameters>  
  
  <Syntax: types>  
  
  <Syntax: interfaces>  
  
contains  
  
  <Syntax: subroutines>  
  
end module syntax_rules
```

#### 3.3.1 Syntax rules

Syntax rules are used by the parser. They determine how to translate the stream of lexemes as returned by the lexer into the parse tree node. A rule may be terminal, i.e., replace a matching lexeme into a terminal node. The node will contain the lexeme interpreted as a recognized token:

- a keyword: unquoted fixed character string;
- a real number, to be determined at runtime;
- an integer, to be determined at runtime;
- a boolean value, to be determined at runtime;
- a quoted token (e.g., string), to be determined at runtime;
- an identifier (unquoted string that is not a recognized keyword), to be determined at runtime.

It may be nonterminal, i.e., contain a sequence of child rules. These are matched consecutively (and recursively) against the input stream; the resulting node will be a branch node.

- the file, i.e., the input stream as a whole;
- a sequence of syntax elements, where the last syntax element may be optional, or optional repetitive;

Sequences carry a flag that tells whether the last child is optional or may be repeated an arbitrary number of times, corresponding to the regexp modifiers `?`, `*`, and `+`.

We also need an alternative rule; this will be replaced by the node generated by one of its children that matches; thus, it does not create a node of its own.

- an alternative of syntax elements.

We also define special types of sequences as convenience macros:

- a list: a sequence where the elements are separated by a separator keyword (e.g., commas), the separators are thrown away when parsing the list;
- a group: a sequence of three tokens, where the first and third ones are left and right delimiters, the delimiters are thrown away;
- an argument list: a delimited list, containing both delimiters and separators.

It would be great to have a polymorphic type for this purpose, but until Fortran 2003 is out we have to emulate this.

Here are the syntax element codes:

```

<Syntax: public>≡
  public :: S_UNKNOWN
  public :: S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_QUOTED
  public :: S_IDENTIFIER, S_KEYWORD
  public :: S_SEQUENCE, S_LIST, S_GROUP, S_ARGS
  public :: S_ALTERNATIVE
  public :: S_IGNORE

<Syntax: parameters>≡
  integer, parameter :: &
    S_UNKNOWN = 0, &
    S_LOGICAL = 1, S_INTEGER = 2, S_REAL = 3, S_COMPLEX = 4, &
    S_QUOTED = 5, S_IDENTIFIER = 6, S_KEYWORD = 7, &
    S_SEQUENCE = 8, S_LIST = 9, S_GROUP = 10, S_ARGS = 11, &
    S_ALTERNATIVE = 12, &
    S_IGNORE = 99

```

We need arrays of rule pointers, therefore this construct.

```

<Syntax: types>≡
  type :: rule_p
  private
    type(syntax_rule_t), pointer :: p => null ()
  end type rule_p

```



Return the association status of the rule pointer:

```

<Syntax: subroutines>≡
    elemental function rule_is_associated (rp) result (ok)
        logical :: ok
        type (rule_p), intent(in) :: rp
        ok = associated (rp%p)
    end function rule_is_associated

```

The rule type is one of the types listed above, represented by an integer code. The keyword, for a non-keyword rule, is an identifier used for the printed syntax table. The array of children is needed for nonterminal rules. In that case, there is a modifier for the last element (blank, "?", "\*", or "+"), mirrored in the flags **opt** and **rep**. Then, we have the character constants used as separators and delimiters for this rule. Finally, the **used** flag can be set to indicate that this rule is the child of another rule.

```

<Syntax: types>+≡
    public :: syntax_rule_t

<Syntax: types>+≡
    type :: syntax_rule_t
        private
        integer :: type = S_UNKNOWN
        logical :: used = .false.
        type(string_t) :: keyword
        type(string_t) :: separator
        type(string_t), dimension(2) :: delimiter
        type(rule_p), dimension(:), allocatable :: child
        character(1) :: modifier = ""
        logical :: opt = .false., rep = .false.
    end type syntax_rule_t

```

Initializer: Set type and key for a rule, but do not (yet) allocate anything.

Finalizer: not needed (no pointer components).

```

<Syntax: subroutines>+≡
    subroutine syntax_rule_init (rule, key, type)
        type(syntax_rule_t), intent(inout) :: rule
        type(string_t), intent(in) :: key
        integer, intent(in) :: type
        rule%keyword = key
        rule%type = type
        select case (rule%type)
            case (S_GROUP)
                call syntax_rule_set_delimiter (rule)
            case (S_LIST)
                call syntax_rule_set_separator (rule)
            case (S_ARGS)
                call syntax_rule_set_delimiter (rule)
                call syntax_rule_set_separator (rule)
        end select
    end subroutine syntax_rule_init

```

### 3.3.2 I/O

These characters will not be enclosed in quotes when writing syntax rules:

```
<Limits: public parameters>+≡
    character(*), parameter, public :: &
        UNQUOTED = "(),|_"/LCLETTERS/UCLETTERS/DIGITS
```

Write an account of the rule. Setting `short` true will suppress the node type. Setting `key_only` true will suppress the definition. Setting `advance` false will suppress the trailing newline.

```
<Syntax: public>+≡
    public :: syntax_rule_write

<Syntax: subroutines>+≡
    subroutine syntax_rule_write (rule, unit, short, key_only, advance)
        type(syntax_rule_t), intent(in) :: rule
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: short, key_only, advance
        logical :: typ, def, adv
        integer :: u
        u = output_unit (unit); if (u < 0) return
        typ = .true.; if (present (short)) typ = .not. short
        def = .true.; if (present (key_only)) def = .not. key_only
        adv = .true.; if (present (advance)) adv = advance
        select case (rule%type)
        case (S_UNKNOWN); call write_atom ("???", typ)
        case (S_IGNORE); call write_atom ("IGNORE", typ)
        case (S_LOGICAL); call write_atom ("LOGICAL", typ)
        case (S_INTEGER); call write_atom ("INTEGER", typ)
        case (S_REAL); call write_atom ("REAL", typ)
        case (S_COMPLEX); call write_atom ("COMPLEX", typ)
        case (S_IDENTIFIER); call write_atom ("IDENTIFIER", typ)
        case (S_KEYWORD); call write_atom ("KEYWORD", typ)
        case (S_QUOTED)
            call write_quotes (typ, def, del = rule%delimiter)
        case (S_SEQUENCE)
            call write_sequence ("SEQUENCE", typ, def, size (rule%child))
        case (S_GROUP)
            call write_sequence ("GROUP", typ, def, size (rule%child), &
                del = rule%delimiter)
        case (S_LIST)
            call write_sequence ("LIST", typ, def, size (rule%child), &
                sep = rule%separator)
        case (S_ARGS)
            call write_sequence ("ARGUMENTS", typ, def, size (rule%child), &
                del = rule%delimiter, sep = rule%separator)
        case (S_ALTERNATIVE)
            call write_sequence ("ALTERNATIVE", typ, def, size (rule%child), &
                sep = var_str ("|"))
        end select
        if (adv) write (u, *)
contains
        subroutine write_type (type)
            character(*), intent(in) :: type
            character(11) :: str
```

```

    str = type
    write (u, "(1x,A)", advance="no") str
end subroutine write_type
subroutine write_key
    write (u, "(1x,A)", advance="no") char (wkey (rule))
end subroutine write_key
subroutine write_atom (type, typ)
    character(*), intent(in) :: type
    logical, intent(in) :: typ
    if (typ) call write_type (type)
    call write_key
end subroutine write_atom
subroutine write_maybe_quoted (string)
    character(*), intent(in) :: string
    character, parameter :: q = '"'
    character, parameter :: qq = '''
    if (verify (string, UNQUOTED) == 0) then
        write (u, "(1x,A)", advance = "no") trim (string)
    else if (verify (string, q) == 0) then
        write (u, "(1x,A)", advance = "no") qq // trim (string) // qq
    else
        write (u, "(1x,A)", advance = "no") q // trim (string) // q
    end if
end subroutine write_maybe_quoted
subroutine write_quotes (typ, def, del)
    logical, intent(in) :: typ, def
    type(string_t), dimension(2), intent(in) :: del
    if (typ) call write_type ("QUOTED")
    call write_key
    if (def) then
        write (u, "(1x, '=')", advance="no")
        call write_maybe_quoted (char (del(1)))
        write (u, "(1x,A)", advance="no") "..."
        call write_maybe_quoted (char (del(2)))
    end if
end subroutine write_quotes
subroutine write_sequence (type, typ, def, n, del, sep)
    character(*), intent(in) :: type
    logical, intent(in) :: typ, def
    integer, intent(in) :: n
    type(string_t), dimension(2), intent(in), optional :: del
    type(string_t), intent(in), optional :: sep
    integer :: i
    if (typ) call write_type (type)
    call write_key
    if (def) then
        write (u, "(1x, '=')", advance="no")
        if (present (del)) call write_maybe_quoted (char (del(1)))
        do i = 1, n
            if (i > 1 .and. present (sep)) &
                call write_maybe_quoted (char (sep))
            write (u, "(1x,A)", advance="no") &
                char (wkey (syntax_rule_get_sub_ptr(rule, i)))
            if (i == n) write (u, "(A)", advance="no") trim (rule%modifier)
        end do
    end if
end subroutine write_sequence

```

```

        end do
        if (present (del)) call write_maybe_quoted (char (del(2)))
        end if
    end subroutine write_sequence
end subroutine syntax_rule_write

```

In the printed representation, the keyword strings are enclosed as `<...>`, unless they are bare keywords. Bare keywords are enclosed as `'..'` if they contain a character which is not a letter, digit, or underscore. If they contain a single-quote character, they are enclosed as `".."`. (A keyword must not contain both single- and double-quotes.)

```

<Syntax: subroutines>+≡
function wkey (rule) result (string)
    type(string_t) :: string
    type(syntax_rule_t), intent(in) :: rule
    select case (rule%type)
    case (S_KEYWORD)
        if (verify (rule%keyword, UNQUOTED) == 0) then
            string = rule%keyword
        else if (scan (rule%keyword, "'") == 0) then
            string = "'" // rule%keyword // "'"
        else
            string = '""' // rule%keyword // '""'
        end if
    case default
        string = "<" // rule%keyword // ">"
    end select
end function wkey

```

### 3.3.3 Completing syntax rules

Set the separator and delimiter entries, using defaults:

```

<Syntax: subroutines>+≡
subroutine syntax_rule_set_separator (rule, separator)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), intent(in), optional :: separator
    if (present (separator)) then
        rule%separator = separator
    else
        rule%separator = ","
    end if
end subroutine syntax_rule_set_separator

subroutine syntax_rule_set_delimiter (rule, delimiter)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), dimension(2), intent(in), optional :: delimiter
    if (present (delimiter)) then
        rule%delimiter = delimiter
    else
        rule%delimiter = (/ "(", ")" /)
    end if
end subroutine syntax_rule_set_delimiter

```

```
end subroutine syntax_rule_set_delimiter
```

Set the modifier entry and corresponding flags:

*<Syntax: subroutines>+≡*

```
function is_modifier (string) result (ok)
  logical :: ok
  type(string_t), intent(in) :: string
  select case (char (string))
    case (" ", "?", "*", "+"); ok = .true.
    case default; ok = .false.
  end select
end function is_modifier

subroutine syntax_rule_set_modifier (rule, modifier)
  type(syntax_rule_t), intent(inout) :: rule
  type(string_t), intent(in) :: modifier
  rule%modifier = char (modifier)
  select case (rule%modifier)
    case (" ")
    case ("?"); rule%opt = .true.
    case ("*"); rule%opt = .true.; rule%rep = .true.
    case ("+"); rule%rep = .true.
    case default
      call msg_bug (" Syntax: sequence modifier '" // rule%modifier &
        // "' is not one of '+' '*' '?'")
  end select
end subroutine syntax_rule_set_modifier
```

Check a finalized rule for completeness

*<Syntax: subroutines>+≡*

```
subroutine syntax_rule_check (rule)
  type(syntax_rule_t), intent(in) :: rule
  if (rule%keyword == "") call msg_bug ("Rule key not set")
  select case (rule%type)
    case (S_UNKNOWN); call bug (" Undefined rule")
    case (S_IGNORE, S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, &
      S_IDENTIFIER, S_KEYWORD)
    case (S_QUOTED)
      if (any (rule%delimiter == "")) call bug (" Missing quote character(s)")
    case (S_SEQUENCE)
    case (S_GROUP)
      if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
    case (S_LIST)
      if (rule%separator == "") call bug (" Missing separator")
    case (S_ARGS)
      if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
      if (rule%separator == "") call bug (" Missing separator")
    case (S_ALTERNATIVE)
    case default
      call bug (" Undefined syntax code")
  end select
  select case (rule%type)
    case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
```

```

        if (allocated (rule%child)) then
            if (.not.all (rule_is_associated (rule%child))) &
                call bug (" Child rules not all associated")
        else
            call bug (" Parent rule without children")
        end if
    case default
        if (allocated (rule%child)) call bug (" Non-parent rule with children")
    end select
contains
    subroutine bug (string)
        character(*), intent(in) :: string
        call msg_bug (" Syntax table: Rule " // char (rule%keyword) // ": " &
            // string)
    end subroutine bug
end subroutine syntax_rule_check

```

### 3.3.4 Accessing rules

This is the API for syntax rules:

```

<Syntax: public>+≡
    public :: syntax_rule_get_type

<Syntax: subroutines>+≡
    function syntax_rule_get_type (rule) result (type)
        integer :: type
        type(syntax_rule_t), intent(in) :: rule
        type = rule%type
    end function syntax_rule_get_type

<Syntax: public>+≡
    public :: syntax_rule_get_key

<Syntax: subroutines>+≡
    function syntax_rule_get_key (rule) result (key)
        type(string_t) :: key
        type(syntax_rule_t), intent(in) :: rule
        key = rule%keyword
    end function syntax_rule_get_key

<Syntax: public>+≡
    public :: syntax_rule_get_separator
    public :: syntax_rule_get_delimiter

<Syntax: subroutines>+≡
    function syntax_rule_get_separator (rule) result (separator)
        type(string_t) :: separator
        type(syntax_rule_t), intent(in) :: rule
        separator = rule%separator
    end function syntax_rule_get_separator

    function syntax_rule_get_delimiter (rule) result (delimiter)
        type(string_t), dimension(2) :: delimiter

```

```

    type(syntax_rule_t), intent(in) :: rule
    delimiter = rule%delimiter
end function syntax_rule_get_delimiter

```

Accessing child rules. If we use `syntax_rule_get_n_sub` for determining loop bounds, we do not need a check in the second routine.

```

<Syntax: public>+≡
    public :: syntax_rule_get_n_sub
    public :: syntax_rule_get_sub_ptr

<Syntax: subroutines>+≡
    function syntax_rule_get_n_sub (rule) result (n)
        integer :: n
        type(syntax_rule_t), intent(in) :: rule
        if (allocated (rule%child)) then
            n = size (rule%child)
        else
            n = 0
        end if
    end function syntax_rule_get_n_sub

    function syntax_rule_get_sub_ptr (rule, i) result (sub)
        type(syntax_rule_t), pointer :: sub
        type(syntax_rule_t), intent(in), target :: rule
        integer, intent(in) :: i
        sub => rule%child(i)%p
    end function syntax_rule_get_sub_ptr

    subroutine syntax_rule_set_sub (rule, i, sub)
        type(syntax_rule_t), intent(inout) :: rule
        integer, intent(in) :: i
        type(syntax_rule_t), intent(in), target :: sub
        rule%child(i)%p => sub
    end subroutine syntax_rule_set_sub

```

Return the modifier flags:

```

<Syntax: public>+≡
    public :: syntax_rule_last_optional
    public :: syntax_rule_last_repetitive

<Syntax: subroutines>+≡
    function syntax_rule_last_optional (rule) result (opt)
        logical :: opt
        type(syntax_rule_t), intent(in) :: rule
        opt = rule%opt
    end function syntax_rule_last_optional
    function syntax_rule_last_repetitive (rule) result (rep)
        logical :: rep
        type(syntax_rule_t), intent(in) :: rule
        rep = rule%rep
    end function syntax_rule_last_repetitive

```

Return true if the rule is atomic, i.e., logical, real, keyword etc.

```
<Syntax: public>+≡
    public :: syntax_rule_is_atomic

<Syntax: subroutines>+≡
    function syntax_rule_is_atomic (rule) result (atomic)
        logical :: atomic
        type(syntax_rule_t), intent(in) :: rule
        select case (rule%type)
            case (S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_IDENTIFIER, &
                 S_KEYWORD, S_QUOTED)
                atomic = .true.
            case default
                atomic = .false.
        end select
    end function syntax_rule_is_atomic
```

### 3.3.5 Syntax tables

A syntax table contains the tree of syntax rules and, for direct parser access, the list of valid keywords.

#### Types

The syntax contains an array of rules and a list of keywords. The array is actually used as a tree, where the top rule is the first array element, and the other rules are recursively pointed to by this first rule. (No rule should be used twice or be unused.) The keyword list is derived from the rule tree.

Objects of this type need the target attribute if they are associated with a lexer. The keyword list will be pointed to by this lexer.

```
<Syntax: public>+≡
    public :: syntax_t

<Syntax: types>+≡
    type :: syntax_t
        private
        type(syntax_rule_t), dimension(:), allocatable :: rule
        type(keyword_list_t) :: keyword_list
    end type syntax_t
```

#### Constructor/destructor

Initialize and finalize syntax tables

```
<Syntax: public>+≡
    public :: syntax_init
    public :: syntax_final
```



There are two ways to create a syntax: hard-coded from rules or dynamically from file.

```

<Syntax: interfaces>≡
interface syntax_init
  module procedure syntax_init_from_ifile
end interface

```

The syntax definition is read from an ifile object which contains the syntax definitions in textual form, one rule per line. This interface allows for determining the number of rules beforehand.

To parse the rule definitions, we make up a temporary lexer. Obviously, we cannot use a generic parser yet, so we have to hardcode the parsing process.

```

<Syntax: subroutines>+≡
subroutine syntax_init_from_ifile (syntax, ifile)
  type(syntax_t), intent(out), target :: syntax
  type(ifile_t), intent(in) :: ifile
  type(lexer_t) :: lexer
  type(line_p) :: line
  type(string_t) :: string
  integer :: n_token
  integer :: i
  call lexer_init (lexer, &
    comment_chars = "", &
    quote_chars = "<'\"", &
    quote_match = ">'\"", &
    single_chars = "?*+|=,()", &
    special_class = (/ "." /), &
    keyword_list = null ())
  allocate (syntax%rule (ifile_get_length (ifile)))
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
    string = line_get_string_advance (line)
    call set_rule_type_and_key (syntax%rule(i), string, lexer)
  end do
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
    string = line_get_string_advance (line)
    select case (syntax%rule(i)%type)
    case (S_QUOTED, S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
      n_token = get_n_token (string, lexer)
      call set_rule_contents &
        (syntax%rule(i), syntax, n_token, string, lexer)
    end select
  end do
  call line_final (line)
  call lexer_final (lexer)
  call syntax_make_keyword_list (syntax)
  if (.not. all (syntax%rule%used)) then
    do i = 1, size (syntax%rule)
      if (.not. syntax%rule(i)%used) then
        call syntax_rule_write (syntax%rule(i), 6)
      end if
    end do
  end do
end do

```

```

        call msg_bug (" Syntax table: unused rules")
    end if
end subroutine syntax_init_from_ifile

```

For a given rule defined in the input, the first task is to determine its type and key. With these, we can initialize the rule in the table, postponing the association of children.

```

<Syntax: subroutines>+=
subroutine set_rule_type_and_key (rule, string, lexer)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t) :: lexeme
    type(string_t) :: key
    character(2) :: type
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    call lex (lexeme, lexer)
    type = lexeme_get_string (lexeme)
    call lex (lexeme, lexer)
    key = lexeme_get_contents (lexeme)
    call stream_final (stream)
    if (trim (key) /= "") then
        select case (type)
            case ("IG"); call syntax_rule_init (rule, key, S_IGNORE)
            case ("LO"); call syntax_rule_init (rule, key, S_LOGICAL)
            case ("IN"); call syntax_rule_init (rule, key, S_INTEGER)
            case ("RE"); call syntax_rule_init (rule, key, S_REAL)
            case ("CO"); call syntax_rule_init (rule, key, S_COMPLEX)
            case ("ID"); call syntax_rule_init (rule, key, S_IDENTIFIER)
            case ("KE"); call syntax_rule_init (rule, key, S_KEYWORD)
            case ("QU"); call syntax_rule_init (rule, key, S_QUOTED)
            case ("SE"); call syntax_rule_init (rule, key, S_SEQUENCE)
            case ("GR"); call syntax_rule_init (rule, key, S_GROUP)
            case ("LI"); call syntax_rule_init (rule, key, S_LIST)
            case ("AR"); call syntax_rule_init (rule, key, S_ARGS)
            case ("AL"); call syntax_rule_init (rule, key, S_ALTERNATIVE)
            case default
                call lexer_show_location (lexer)
                call msg_bug (" Syntax definition: unknown type ' " // type // "'")
            end select
        else
            print *, char (string)
            call msg_bug (" Syntax definition: empty rule key")
        end if
    end subroutine set_rule_type_and_key

```

This function returns the number of tokens in an input line.

```

<Syntax: subroutines>+=
function get_n_token (string, lexer) result (n)
    integer :: n

```

```

type(string_t), intent(in) :: string
type(lexer_t), intent(inout) :: lexer
type(stream_t), target :: stream
type(lexeme_t) :: lexeme
integer :: i
call lexer_clear (lexer)
call stream_init (stream, string)
call lexer_assign_stream (lexer, stream)
i = 0
do
    call lex (lexeme, lexer)
    if (lexeme_is_break (lexeme)) exit
    i = i + 1
end do
n = i
call stream_final (stream)
end function get_n_token

```

This subroutine extracts the rule contents for an input line. There are three tasks: (1) determine the number of children, depending on the rule type; (2) find and set the separator and delimiter strings, if required; (3) scan the child rules, find them in the syntax table and associate the parent rule with them.

(*Syntax: subroutines*)+≡

```

subroutine set_rule_contents (rule, syntax, n_token, string, lexer)
    type(syntax_rule_t), intent(inout) :: rule
    type(syntax_t), intent(in), target :: syntax
    integer, intent(in) :: n_token
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t), dimension(n_token) :: lexeme
    integer :: i, n_children
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    do i = 1, n_token
        call lex (lexeme(i), lexer)
    end do
    call stream_final (stream)
    n_children = get_n_children ()
    call set_delimiters
    if (n_children > 1) call set_separator
    if (n_children > 0) call set_children
contains
    function get_n_children () result (n_children)
        integer :: n_children
        select case (rule%type)
        case (S_QUOTED)
            if (n_token /= 6) call broken_rule (rule)
            n_children = 0
        case (S_GROUP)
            if (n_token /= 6) call broken_rule (rule)
            n_children = 1

```

```

case (S_SEQUENCE)
  if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
    if (n_token <= 4) call broken_rule (rule)
    call syntax_rule_set_modifier &
      (rule, lexeme_get_string (lexeme(n_token)))
    n_children = n_token - 4
  else
    if (n_token <= 3) call broken_rule (rule)
    n_children = n_token - 3
  end if
case (S_LIST)
  if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
    if (n_token <= 4 .or. mod (n_token, 2) /= 1) &
      call broken_rule (rule)
    call syntax_rule_set_modifier &
      (rule, lexeme_get_string (lexeme(n_token)))
  else if (n_token <= 3 .or. mod (n_token, 2) /= 0) then
    call broken_rule (rule)
  end if
  n_children = (n_token - 2) / 2
case (S_ARGS)
  if (is_modifier (lexeme_get_string (lexeme(n_token-1)))) then
    if (n_token <= 6 .or. mod (n_token, 2) /= 1) &
      call broken_rule (rule)
    call syntax_rule_set_modifier &
      (rule, lexeme_get_string (lexeme(n_token-1)))
  else if (n_token <= 5 .or. mod (n_token, 2) /= 0) then
    call broken_rule (rule)
  end if
  n_children = (n_token - 4) / 2
case (S_ALTERNATIVE)
  if (n_token <= 3 .or. mod (n_token, 2) /= 0) call broken_rule (rule)
  n_children = (n_token - 2) / 2
end select
end function get_n_children
subroutine set_delimiters
  type(string_t), dimension(2) :: delimiter
  select case (rule%type)
  case (S_QUOTED, S_GROUP, S_ARGS)
    delimiter(1) = lexeme_get_contents (lexeme(4))
    delimiter(2) = lexeme_get_contents (lexeme(n_token))
    call syntax_rule_set_delimiter (rule, delimiter)
  end select
end subroutine set_delimiters
subroutine set_separator
  type(string_t) :: separator
  select case (rule%type)
  case (S_LIST)
    separator = lexeme_get_contents (lexeme(5))
    call syntax_rule_set_separator (rule, separator)
  case (S_ARGS)
    separator = lexeme_get_contents (lexeme(6))
    call syntax_rule_set_separator (rule, separator)
  end select
end subroutine set_separator

```

```

end subroutine set_separator
subroutine set_children
  allocate (rule%child(n_children))
  select case (rule%type)
  case (S_GROUP)
    call syntax_rule_set_sub (rule, 1, syntax_get_rule_ptr (syntax, &
      lexeme_get_contents (lexeme(5))))
  case (S_SEQUENCE)
    do i = 1, n_children
      call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
        lexeme_get_contents (lexeme(i+3))))
    end do
  case (S_LIST, S_ALTERNATIVE)
    do i = 1, n_children
      call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
        lexeme_get_contents (lexeme(2*i+2))))
    end do
  case (S_ARGS)
    do i = 1, n_children
      call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
        lexeme_get_contents (lexeme(2*i+3))))
    end do
  end select
end subroutine set_children
subroutine broken_rule (rule)
  type(syntax_rule_t), intent(in) :: rule
  call lexer_show_location (lexer)
  call msg_bug (" Syntax definition: broken rule '" &
    // char (wkey (rule)) // "'")
end subroutine broken_rule
end subroutine set_rule_contents

```

This routine completes the syntax table object. We assume that the rule array is set up. We associate the top rule with the first entry in the rule array and build up the keyword list.

The keyword list includes delimiters and separators. Filling it can only be done after all rules are set. We scan the rule tree. For each keyword that we find, we try to add it to the keyword list; the pointer to the last element is carried along with the recursive scanning. Before appending a keyword, we check whether it is already in the list.

*(Syntax: subroutines)*+≡

```

subroutine syntax_make_keyword_list (syntax)
  type(syntax_t), intent(inout), target :: syntax
  type(syntax_rule_t), pointer :: rule
  rule => syntax%rule(1)
  call rule_scan_rec (rule, syntax%keyword_list)
contains
  recursive subroutine rule_scan_rec (rule, keyword_list)
    type(syntax_rule_t), pointer :: rule
    type(keyword_list_t), intent(inout) :: keyword_list
    integer :: i
    if (rule%used) return
    rule%used = .true.

```

```

select case (rule%type)
case (S_UNKNOWN)
  call msg_bug (" Syntax: rule tree contains undefined rule")
case (S_KEYWORD)
  call keyword_list_add (keyword_list, rule%keyword)
end select
select case (rule%type)
case (S_LIST, S_ARGS)
  call keyword_list_add (keyword_list, rule%separator)
end select
select case (rule%type)
case (S_GROUP, S_ARGS)
  call keyword_list_add (keyword_list, rule%delimiter(1))
  call keyword_list_add (keyword_list, rule%delimiter(2))
end select
select case (rule%type)
case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
  if (.not. allocated (rule%child)) &
    call msg_bug (" Syntax: Non-terminal rule without children")
case default
  if (allocated (rule%child)) &
    call msg_bug (" Syntax: Terminal rule with children")
end select
if (allocated (rule%child)) then
  do i = 1, size (rule%child)
    call rule_scan_rec (rule%child(i)%p, keyword_list)
  end do
end if
end subroutine rule_scan_rec
end subroutine syntax_make_keyword_list

```

The finalizer deallocates the rule pointer array and deletes the keyword list.

```

<Syntax: subroutines>+≡
subroutine syntax_final (syntax)
  type(syntax_t), intent(inout) :: syntax
  if (allocated (syntax%rule)) deallocate (syntax%rule)
  call keyword_list_final (syntax%keyword_list)
end subroutine syntax_final

```

### 3.3.6 Accessing the syntax table

Return a pointer to the top rule

```

<Syntax: public>+≡
public :: syntax_get_top_rule_ptr

<Syntax: subroutines>+≡
function syntax_get_top_rule_ptr (syntax) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(syntax_t), intent(in), target :: syntax
  if (allocated (syntax%rule)) then
    rule => syntax%rule(1)
  else

```

```

        rule => null ()
    end if
end function syntax_get_top_rule_ptr

```

Assign the pointer to the rule associated with a given key (assumes that the rule array is allocated)

```

<Syntax: public>+≡
    public :: syntax_get_rule_ptr

<Syntax: subroutines>+≡
    function syntax_get_rule_ptr (syntax, key) result (rule)
        type(syntax_rule_t), pointer :: rule
        type(syntax_t), intent(in), target :: syntax
        type(string_t), intent(in) :: key
        integer :: i
        do i = 1, size (syntax%rule)
            if (syntax%rule(i)%keyword == key) then
                rule => syntax%rule(i)
                return
            end if
        end do
        call msg_bug (" Syntax table: Rule " // char (key) // " not found")
    end function syntax_get_rule_ptr

```

Return a pointer to the keyword list

```

<Syntax: public>+≡
    public :: syntax_get_keyword_list_ptr

<Syntax: subroutines>+≡
    function syntax_get_keyword_list_ptr (syntax) result (keyword_list)
        type(keyword_list_t), pointer :: keyword_list
        type(syntax_t), intent(in), target :: syntax
        keyword_list => syntax%keyword_list
    end function syntax_get_keyword_list_ptr

```

### 3.3.7 I/O

Write a readable representation of the syntax table

```

<Syntax: public>+≡
    public :: syntax_write

<Syntax: subroutines>+≡
    subroutine syntax_write (syntax, unit)
        type(syntax_t), intent(in) :: syntax
        integer, intent(in), optional :: unit
        integer :: u
        integer :: i
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)") "Syntax table:"
        if (allocated (syntax%rule)) then
            do i = 1, size (syntax%rule)
                call syntax_rule_write (syntax%rule(i), u)
            end do
        end if
    end subroutine syntax_write

```

```
        end do
    else
        write (u, "(1x,A)") "[not allocated]"
    end if
    call keyword_list_write (syntax%keyword_list, u)
end subroutine syntax_write
```



## 3.4 The parser

On a small scale, the parser interprets the string tokens returned by the lexer; they are interpreted as numbers, keywords and such and stored as a typed object. On a large scale, a text is read, parsed, and a syntax rule set is applied such that the tokens are stored as a parse tree. Syntax errors are spotted in this process, so the resulting parse tree is syntactically correct by definition.

```
<parser.f90>≡
  <File header>

  module parser

    <Use kinds>
    <Use strings>
    use limits, only: DIGITS !NODEP!
    <Use file utils>
    use diagnostics !NODEP!
    use md5
    use lexers
    use syntax_rules

    <Standard module head>

    <Parser: public>

    <Parser: types>

    <Parser: interfaces>

    contains

    <Parser: procedures>

  end module parser
```

### 3.4.1 The token type

Tokens are elements of the parsed input that carry a value: logical, integer, real, quoted string, (unquoted) identifier, or known keyword. Note that non-keyword tokens also have an abstract key attached to them.

This is an obvious candidate for polymorphism.

```
<Parser: types>≡
  type :: token_t
    private
    integer :: type = S_UNKNOWN
    logical, pointer :: lval => null ()
    integer, pointer :: ival => null ()
    real(default), pointer :: rval => null ()
    complex(default), pointer :: cval => null ()
    type(string_t), pointer :: sval => null ()
    type(string_t), pointer :: kval => null ()
    type(string_t), dimension(:), pointer :: quote => null ()
```

```
end type token_t
```

Create a token from the lexeme returned by the lexer: Allocate storage and try to interpret the lexeme according to the type that is requested by the parser. For a keyword token, match the lexeme against the requested key. If successful, set the token type, value, and key. Otherwise, set the type to S\_UNKNOWN.

*(Parser: procedures)*≡

```
subroutine token_init (token, lexeme, requested_type, key)
  type(token_t), intent(out) :: token
  type(lexeme_t), intent(in)  :: lexeme
  integer, intent(in) :: requested_type
  type(string_t), intent(in) :: key
  integer :: type
  type = lexeme_get_type (lexeme)
  token%type = S_UNKNOWN
  select case (requested_type)
  case (S_LOGICAL)
    if (type == T_IDENTIFIER) call read_logical &
      (char (lexeme_get_string (lexeme)))
  case (S_INTEGER)
    if (type == T_NUMERIC) call read_integer &
      (char (lexeme_get_string (lexeme)))
  case (S_REAL)
    if (type == T_NUMERIC) call read_real &
      (char (lexeme_get_string (lexeme)))
  case (S_COMPLEX)
    if (type == T_NUMERIC) call read_complex &
      (char (lexeme_get_string (lexeme)))
  case (S_IDENTIFIER)
    if (type == T_IDENTIFIER) call read_identifier &
      (lexeme_get_string (lexeme))
  case (S_KEYWORD)
    if (type == T_KEYWORD) call check_keyword &
      (lexeme_get_string (lexeme), key)
  case (S_QUOTED)
    if (type == T_QUOTED) call read_quoted &
      (lexeme_get_contents (lexeme), lexeme_get_delimiters (lexeme))
  case default
    print *, requested_type
    call msg_bug (" Invalid token type code requested by the parser")
  end select
  if (token%type /= S_UNKNOWN) then
    allocate (token%kval)
    token%kval = key
  end if
contains
  subroutine read_logical (s)
    character(*), intent(in) :: s
    select case (s)
    case ("t", "T", "true", "TRUE", "y", "Y", "yes", "YES")
      allocate (token%lval)
      token%lval = .true.
      token%type = S_LOGICAL
```

```

        case ("f", "F", "false", "FALSE", "n", "N", "no", "NO")
            allocate (token%lval)
            token%lval = .false.
            token%type = S_LOGICAL
        end select
end subroutine read_logical
subroutine read_integer (s)
    character(*), intent(in) :: s
    integer :: tmp, iostat
    if (verify (s, DIGITS) == 0) then
        read (s, *, iostat=iostat) tmp
        if (iostat == 0) then
            allocate (token%ival)
            token%ival = tmp
            token%type = S_INTEGER
        end if
    end if
end subroutine read_integer
subroutine read_real (s)
    character(*), intent(in) :: s
    real(default) :: tmp
    integer :: iostat
    read (s, *, iostat=iostat) tmp
    if (iostat == 0) then
        allocate (token%rval)
        token%rval = tmp
        token%type = S_REAL
    end if
end subroutine read_real
subroutine read_complex (s)
    character(*), intent(in) :: s
    complex(default) :: tmp
    integer :: iostat
    read (s, *, iostat=iostat) tmp
    if (iostat == 0) then
        allocate (token%cval)
        token%cval = tmp
        token%type = S_COMPLEX
    end if
end subroutine read_complex
subroutine read_identifier (s)
    type(string_t), intent(in) :: s
    allocate (token%sval)
    token%sval = s
    token%type = S_IDENTIFIER
end subroutine read_identifier
subroutine check_keyword (s, key)
    type(string_t), intent(in) :: s
    type(string_t), intent(in) :: key
    if (key == s) token%type = S_KEYWORD
end subroutine check_keyword
subroutine read_quoted (s, del)
    type(string_t), intent(in) :: s
    type(string_t), dimension(2), intent(in) :: del

```

```

        allocate (token%sval, token%quote(2))
        token%sval = s
        token%quote(1) = del(1)
        token%quote(2) = del(2)
        token%type = S_QUOTED
    end subroutine read_quoted
end subroutine token_init

```

Reset a token to an empty state, freeing allocated memory, and deallocate the token itself.

```

(Parser: procedures)+≡
subroutine token_final (token)
    type(token_t), intent(inout) :: token
    token%type = S_UNKNOWN
    if (associated (token%lval)) deallocate (token%lval)
    if (associated (token%ival)) deallocate (token%ival)
    if (associated (token%rval)) deallocate (token%rval)
    if (associated (token%sval)) deallocate (token%sval)
    if (associated (token%kval)) deallocate (token%kval)
    if (associated (token%quote)) deallocate (token%quote)
end subroutine token_final

```

Check for empty=valid token:

```

(Parser: procedures)+≡
function token_is_valid (token) result (valid)
    logical :: valid
    type(token_t), intent(in) :: token
    valid = token%type /= S_UNKNOWN
end function token_is_valid

```

Write the contents of a token.

```

(Parser: procedures)+≡
subroutine token_write (token, unit)
    type(token_t), intent(in) :: token
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    select case (token%type)
    case (S_LOGICAL)
        write (u, "(L1)") token%lval
    case (S_INTEGER)
        write (u, "(I0)") token%ival
    case (S_REAL)
        write (u, "(ES19.12)") token%rval
    case (S_COMPLEX)
        write (u, "('(',ES19.12,',',ES19.12,')')") token%cval
    case (S_IDENTIFIER)
        write (u, "(A)") char (token%sval)
    case (S_KEYWORD)
        write (u, "(A,A)") '[keyword]' // char (token%kval)
    case (S_QUOTED)
        write (u, "(A)") &

```

```

        char (token%quote(1)) // char (token%sval) // char (token%quote(2))
    case default
        write (u, "(A)") '[empty]'
    end select
end subroutine token_write

```

Token assignment via deep copy. This is useful to avoid confusion when the token is transferred to some parse-tree node.

```

(Parser: interfaces)≡
    interface assignment(=)
        module procedure token_assign
    end interface

```

We need to copy only the contents that are actually assigned, the other pointers remain disassociated.

```

(Parser: procedures)+≡
    subroutine token_assign (token, token_in)
        type(token_t), intent(out) :: token
        type(token_t), intent(in) :: token_in
        token%type = token_in%type
        select case (token%type)
            case (S_LOGICAL);    allocate (token%lval); token%lval = token_in%lval
            case (S_INTEGER);    allocate (token%ival); token%ival = token_in%ival
            case (S_REAL);       allocate (token%rval); token%rval = token_in%rval
            case (S_COMPLEX);    allocate (token%cval); token%cval = token_in%cval
            case (S_IDENTIFIER); allocate (token%sval); token%sval = token_in%sval
            case (S_QUOTED);     allocate (token%sval); token%sval = token_in%sval
                                allocate (token%quote(2)); token%quote = token_in%quote
        end select
        if (token%type /= S_UNKNOWN) then
            allocate (token%kval); token%kval = token_in%kval
        end if
    end subroutine token_assign

```

### 3.4.2 Retrieve token contents

These functions all do a trivial sanity check that should avoid crashes.

```

(Parser: procedures)+≡
    function token_get_logical (token) result (lval)
        logical :: lval
        type(token_t), intent(in) :: token
        if (associated (token%lval)) then
            lval = token%lval
        else
            call token_mismatch (token, "logical")
        end if
    end function token_get_logical

    function token_get_integer (token) result (ival)
        integer :: ival
        type(token_t), intent(in) :: token

```

```

    if (associated (token%ival)) then
        ival = token%ival
    else
        call token_mismatch (token, "integer")
    end if
end function token_get_integer

function token_get_real (token) result (rval)
    real(default) :: rval
    type(token_t), intent(in) :: token
    if (associated (token%rval)) then
        rval = token%rval
    else
        call token_mismatch (token, "real")
    end if
end function token_get_real

function token_get_cmplx (token) result (cval)
    complex(default) :: cval
    type(token_t), intent(in) :: token
    if (associated (token%cval)) then
        cval = token%cval
    else
        call token_mismatch (token, "complex")
    end if
end function token_get_cmplx

function token_get_string (token) result (sval)
    type(string_t) :: sval
    type(token_t), intent(in) :: token
    if (associated (token%sval)) then
        sval = token%sval
    else
        call token_mismatch (token, "string")
    end if
end function token_get_string

function token_get_key (token) result (kval)
    type(string_t) :: kval
    type(token_t), intent(in) :: token
    if (associated (token%kval)) then
        kval = token%kval
    else
        call token_mismatch (token, "keyword")
    end if
end function token_get_key

function token_get_quote (token) result (quote)
    type(string_t), dimension(2) :: quote
    type(token_t), intent(in) :: token
    if (associated (token%quote)) then
        quote = token%quote
    else
        call token_mismatch (token, "quote")
    end if
end function token_get_quote

```

```

    end if
end function token_get_quote

```

```

<Parser: procedures>+≡
subroutine token_mismatch (token, type)
    type(token_t), intent(in) :: token
    character(*), intent(in) :: type
    write (6, "(A)", advance="no") "Token: "
    call token_write (token)
    call msg_bug (" Token type mismatch; value required as " // type)
end subroutine token_mismatch

```

### 3.4.3 The parse tree: nodes

The parser will generate a parse tree from the input stream. Each node in this parse tree points to the syntax rule that was applied. (Since syntax rules are stored in a pointer-type array within the syntax table, they qualify as targets.) A leaf node contains a token. A branch node has subnodes. The subnodes are stored as a list, so each node also has a `next` pointer.

```

<Parser: public>≡
    public :: parse_node_t

<Parser: types>+≡
    type :: parse_node_t
    private
        type(syntax_rule_t), pointer :: rule => null ()
        type(token_t) :: token
        integer :: n_sub = 0
        type(parse_node_t), pointer :: sub_first => null ()
        type(parse_node_t), pointer :: sub_last => null ()
        type(parse_node_t), pointer :: next => null ()
    contains
        <Parser: parse node: TBP>
    end type parse_node_t

```

Container for parse node pointers, useful for creating pointer arrays:

```

<Parser: public>+≡
    public :: parse_node_p

<Parser: types>+≡
    type :: parse_node_p
        type(parse_node_t), pointer :: ptr => null ()
    end type parse_node_p

```

Output. The first version writes a node together with its sub-node tree, organized by indentation.

```

<Parser: parse node: TBP>≡
    procedure :: write => parse_node_write_rec

<Parser: public>+≡
    public :: parse_node_write_rec

```

```

(Parser: procedures)+≡
recursive subroutine parse_node_write_rec (node, unit, short, depth)
  class(parse_node_t), intent(in), target :: node
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: short
  integer, intent(in), optional :: depth
  integer :: u, d
  type(parse_node_t), pointer :: current
  u = output_unit (unit); if (u < 0) return
  d = 0; if (present (depth)) d = depth
  call parse_node_write (node, u, short=short)
  current => node%sub_first
  do while (associated (current))
    write (u, "(A)", advance = "no") repeat ("| ", d)
    call parse_node_write_rec (current, unit, short, d+1)
    current => current%next
  end do
end subroutine parse_node_write_rec

```

This does the actual output for a single node, without recursion.

```

(Parser: public)+≡
public :: parse_node_write

(Parser: procedures)+≡
subroutine parse_node_write (node, unit, short)
  class(parse_node_t), intent(in) :: node
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: short
  integer :: u
  type(parse_node_t), pointer :: current
  u = output_unit (unit); if (u < 0) return
  write (u, "(' + ')", advance = "no")
  if (associated (node%rule)) then
    call syntax_rule_write (node%rule, u, &
      short=short, key_only=.true., advance=.false.)
    if (token_is_valid (node%token)) then
      write (u, "(' = ')", advance="no")
      call token_write (node%token, u)
    else if (associated (node%sub_first)) then
      write (u, "(' = ')", advance="no")
      current => node%sub_first
      do while (associated (current))
        call syntax_rule_write (current%rule, u, &
          short=.true., key_only=.true., advance=.false.)
        current => current%next
      end do
      write (u, *)
    else
      write (u, *)
    end if
  else
    write (u, *) "[empty]"
  end if
end subroutine parse_node_write

```



Finalize the token and recursively finalize and deallocate all sub-nodes.

```

(Parser: public)+≡
    public :: parse_node_final

(Parser: procedures)+≡
    recursive subroutine parse_node_final (node, recursive)
        type(parse_node_t), intent(inout) :: node
        type(parse_node_t), pointer :: current
        logical, intent(in), optional :: recursive
        logical :: rec
        rec = .true.; if (present (recursive)) rec = recursive
        call token_final (node%token)
        if (rec) then
            do while (associated (node%sub_first))
                current => node%sub_first
                node%sub_first => node%sub_first%next
                call parse_node_final (current)
                deallocate (current)
            end do
        end if
    end subroutine parse_node_final

```

### 3.4.4 Filling nodes

The constructors allocate and initialize the node. There are two possible initializers (in a later version, should correspond to different type extensions).

First, leaf (terminal) nodes. The token constructor does the actual work, looking at the requested type and key for the given rule and matching against the lexeme contents. If it fails, the token will keep the type `S_UNKNOWN` and remain empty. Otherwise, we have a valid node which contains the new token.

If the lexeme argument is absent, the token is left empty.

```

(Parser: procedures)+≡
    subroutine parse_node_create_leaf (node, rule, lexeme)
        type(parse_node_t), pointer :: node
        type(syntax_rule_t), intent(in), target :: rule
        type(lexeme_t), intent(in) :: lexeme
        allocate (node)
        node%rule => rule
        call token_init (node%token, lexeme, &
            syntax_rule_get_type (rule), syntax_rule_get_key (rule))
        if (.not. token_is_valid (node%token)) deallocate (node)
    end subroutine parse_node_create_leaf

```

Second, branch nodes. We first assign the rule:

```

(Parser: public)+≡
    public :: parse_node_create_branch

(Parser: procedures)+≡
    subroutine parse_node_create_branch (node, rule)
        type(parse_node_t), pointer :: node

```

```

    type(syntax_rule_t), intent(in), target :: rule
    allocate (node)
    node%rule => rule
end subroutine parse_node_create_branch

```

Append a sub-node. The sub-node must exist and be a valid target, otherwise nothing is done.

```

(Parser: public)+≡
    public :: parse_node_append_sub

(Parser: procedures)+≡
    subroutine parse_node_append_sub (node, sub)
        type(parse_node_t), intent(inout) :: node
        type(parse_node_t), pointer :: sub
        if (associated (sub)) then
            if (associated (node%sub_last)) then
                node%sub_last%next => sub
            else
                node%sub_first => sub
            end if
            node%sub_last => sub
        end if
    end subroutine parse_node_append_sub

```

For easy access, once the list is complete we count the number of sub-nodes. If there are no subnodes, the whole node is deleted.

```

(Parser: procedures)+≡
    subroutine parse_node_freeze_branch (node)
        type(parse_node_t), pointer :: node
        type(parse_node_t), pointer :: current
        node%n_sub = 0
        current => node%sub_first
        do while (associated (current))
            node%n_sub = node%n_sub + 1
            current => current%next
        end do
        if (node%n_sub == 0) deallocate (node)
    end subroutine parse_node_freeze_branch

```

Replace the last subnode by a new target. Use with care, this invites to memory mismanagement.

```

(Parser: public)+≡
    public :: parse_node_replace_last_sub

(Parser: procedures)+≡
    subroutine parse_node_replace_last_sub (node, pn_target)
        type(parse_node_t), intent(inout), target :: node
        type(parse_node_t), intent(in), target :: pn_target
        type(parse_node_t), pointer :: current
        integer :: i
        select case (node%n_sub)
        case (1)
            node%sub_first => pn_target

```

```

case (2:)
  current => node%sub_first
  do i = 1, node%n_sub - 2
    current => current%next
  end do
  current%next => pn_target
case default
  call parse_node_write (node)
  call msg_bug ("replace_last_sub' called for non-branch parse node")
end select
node%sub_last => pn_target
end subroutine parse_node_replace_last_sub

```

### 3.4.5 Accessing nodes

Return the node contents. Check if pointers are associated. No check when accessing a sub-node; assume that `parse_node_n_sub` is always used for the upper bound.

The token extractor returns a pointer.

*(Parser: public)+≡*

```

public :: parse_node_get_rule_ptr
public :: parse_node_get_n_sub
public :: parse_node_get_sub_ptr
public :: parse_node_get_next_ptr
public :: parse_node_get_last_sub_ptr

```

*(Parser: procedures)+≡*

```

function parse_node_get_rule_ptr (node) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(parse_node_t), intent(in), target :: node
  if (associated (node%rule)) then
    rule => node%rule
  else
    rule => null ()
    call parse_node_undefined (node, "rule")
  end if
end function parse_node_get_rule_ptr

```

```

function parse_node_get_n_sub (node) result (n)
  integer :: n
  type(parse_node_t), intent(in) :: node
  n = node%n_sub
end function parse_node_get_n_sub

```

```

function parse_node_get_sub_ptr (node, n, tag, required) result (sub)
  type(parse_node_t), pointer :: sub
  type(parse_node_t), intent(in), target :: node
  integer, intent(in), optional :: n
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  integer :: i
  sub => node%sub_first
  if (present (n)) then

```

```

        do i = 2, n
            if (associated (sub)) then
                sub => sub%next
            else
                return
            end if
        end do
    end if
    call parse_node_check (sub, tag, required)
end function parse_node_get_sub_ptr

function parse_node_get_next_ptr (sub, n, tag, required) result (next)
    type(parse_node_t), pointer :: next
    type(parse_node_t), intent(in), target :: sub
    integer, intent(in), optional :: n
    character(*), intent(in), optional :: tag
    logical, intent(in), optional :: required
    integer :: i
    next => sub%next
    if (present (n)) then
        do i = 2, n
            if (associated (next)) then
                next => next%next
            else
                exit
            end if
        end do
    end if
    call parse_node_check (next, tag, required)
end function parse_node_get_next_ptr

function parse_node_get_last_sub_ptr (node, tag, required) result (sub)
    type(parse_node_t), pointer :: sub
    type(parse_node_t), intent(in), target :: node
    character(*), intent(in), optional :: tag
    logical, intent(in), optional :: required
    sub => node%sub_last
    call parse_node_check (sub, tag, required)
end function parse_node_get_last_sub_ptr

<Parser: procedures>+≡
    subroutine parse_node_undefined (node, obj)
        type(parse_node_t), intent(in) :: node
        character(*), intent(in) :: obj
        call parse_node_write (node, 6)
        call msg_bug (" Parse-tree node: " // obj // " requested, but undefined")
    end subroutine parse_node_undefined

Check if a parse node has a particular tag, and if it is associated:

<Parser: public>+≡
    public :: parse_node_check

<Parser: procedures>+≡
    subroutine parse_node_check (node, tag, required)

```

```

type(parse_node_t), pointer :: node
character(*), intent(in), optional :: tag
logical, intent(in), optional :: required
if (associated (node)) then
    if (present (tag)) then
        if (parse_node_get_rule_key (node) /= tag) &
            call parse_node_mismatch (tag, node)
        end if
    else
        if (present (required)) then
            if (required) &
                call msg_bug (" Missing node, expected <" // tag // ">")
            end if
        end if
    end if
end if
end subroutine parse_node_check

```

This is called by a parse-tree scanner if the expected and the actual nodes do not match

```

(Parser: public)+≡
    public :: parse_node_mismatch

(Parser: procedures)+≡
    subroutine parse_node_mismatch (string, parse_node)
        character(*), intent(in) :: string
        type(parse_node_t), intent(in) :: parse_node
        call parse_node_write (parse_node)
        call msg_bug (" Syntax mismatch, expected <" // string // ">.")
    end subroutine parse_node_mismatch

```

The following functions are wrappers for extracting the token contents.

```

(Parser: public)+≡
    public :: parse_node_get_logical
    public :: parse_node_get_integer
    public :: parse_node_get_real
    public :: parse_node_get_cmplx
    public :: parse_node_get_string
    public :: parse_node_get_key
    public :: parse_node_get_rule_key

(Parser: procedures)+≡
    function parse_node_get_logical (node) result (lval)
        logical :: lval
        type(parse_node_t), intent(in), target :: node
        lval = token_get_logical (parse_node_get_token_ptr (node))
    end function parse_node_get_logical

    function parse_node_get_integer (node) result (ival)
        integer :: ival
        type(parse_node_t), intent(in), target :: node
        ival = token_get_integer (parse_node_get_token_ptr (node))
    end function parse_node_get_integer

    function parse_node_get_real (node) result (rval)
        real(default) :: rval

```

```

    type(parse_node_t), intent(in), target :: node
    rval = token_get_real (parse_node_get_token_ptr (node))
end function parse_node_get_real

function parse_node_get_cmplx (node) result (cval)
    complex(default) :: cval
    type(parse_node_t), intent(in), target :: node
    cval = token_get_cmplx (parse_node_get_token_ptr (node))
end function parse_node_get_cmplx

function parse_node_get_string (node) result (sval)
    type(string_t) :: sval
    type(parse_node_t), intent(in), target :: node
    sval = token_get_string (parse_node_get_token_ptr (node))
end function parse_node_get_string

function parse_node_get_key (node) result (kval)
    type(string_t) :: kval
    type(parse_node_t), intent(in), target :: node
    kval = token_get_key (parse_node_get_token_ptr (node))
end function parse_node_get_key

function parse_node_get_rule_key (node) result (kval)
    type(string_t) :: kval
    type(parse_node_t), intent(in), target :: node
    kval = syntax_rule_get_key (parse_node_get_rule_ptr (node))
end function parse_node_get_rule_key

function parse_node_get_token_ptr (node) result (token)
    type(token_t), pointer :: token
    type(parse_node_t), intent(in), target :: node
    if (token_is_valid (node%token)) then
        token => node%token
    else
        call parse_node_undefined (node, "token")
    end if
end function parse_node_get_token_ptr

```

Return a MD5 sum for a parse node. The method is to write the node to a scratch file and to evaluate the MD5 sum of that file.

*<Parser: public>+≡*

```
public :: parse_node_get_md5sum
```

*<Parser: procedures>+≡*

```

function parse_node_get_md5sum (pn) result (md5sum_pn)
    character(32) :: md5sum_pn
    type(parse_node_t), intent(in) :: pn
    integer :: u
    u = free_unit ()
    open (unit = u, status = "scratch", action = "readwrite")
    call parse_node_write_rec (pn, unit=u)
    rewind (u)
    md5sum_pn = md5sum (u)
    close (u)

```

```
end function parse_node_get_md5sum
```

### 3.4.6 The parse tree

The parse tree is a tree of nodes, where leaf nodes hold a valid token, while branch nodes point to a list of sub-nodes.

```
<Parser: public>+≡
    public :: parse_tree_t

<Parser: types>+≡
    type :: parse_tree_t
    private
        type(parse_node_t), pointer :: root_node => null ()
    end type parse_tree_t
```

The parser. Its arguments are the parse tree (which should be empty initially), the lexer (which should be already set up), the syntax table (which should be valid), and the input stream. The input stream is completely parsed, using the lexer setup and the syntax rules as given, and the parse tree is built accordingly.

If `check_eof` is absent or true, the parser will complain about trailing garbage. Otherwise, it will ignore it.

By default, the input stream is matched against the top rule in the specified syntax. If `key` is given, it is matched against the rule with the specified key instead.

Failure at the top level means that no rule could match at all; in this case the error message will show the top rule.

```
<Parser: public>+≡
    public :: parse_tree_init

<Parser: procedures>+≡
    subroutine parse_tree_init &
        (parse_tree, syntax, lexer, key, check_eof)
        type(parse_tree_t), intent(inout) :: parse_tree
        type(lexer_t), intent(inout) :: lexer
        type(syntax_t), intent(in), target :: syntax
        type(string_t), intent(in), optional :: key
        logical, intent(in), optional :: check_eof
        type(syntax_rule_t), pointer :: rule
        type(lexeme_t) :: lexeme
        type(parse_node_t), pointer :: node
        logical :: ok, check
        check = .true.; if (present (check_eof)) check = check_eof
        call lexer_clear (lexer)
        if (present (key)) then
            rule => syntax_get_rule_ptr (syntax, key)
        else
            rule => syntax_get_top_rule_ptr (syntax)
        end if
        if (associated (rule)) then
            call parse_node_match_rule (node, rule, ok)
            if (ok) then
                parse_tree%root_node => node
            end if
        end if
    end subroutine
```

```

else
    call parse_error (rule, lexeme)
end if
if (check) then
    call lex (lexeme, lexer)
    if (.not. lexeme_is_eof (lexeme)) then
        call lexer_show_location (lexer)
        call msg_fatal (" Syntax error " &
            // "(at or before the location indicated above)")
    end if
end if
else
    call msg_bug (" Parser failed because syntax is empty")
end if
contains
<Parser: internal subroutines of parse_tree_init>
end subroutine parse_tree_init

```

The parser works recursively, following the rule tree, building the tree of nodes on the fly. If the given rule matches, the node is filled on return. If not, the node remains empty.

```

<Parser: internal subroutines of parse_tree_init>≡
recursive subroutine parse_node_match_rule (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    logical, parameter :: debug = .false.
    integer :: type
    if (debug) write (6, "(A)", advance="no") "Parsing rule: "
    if (debug) call syntax_rule_write (rule, 6)
    node => null ()
    type = syntax_rule_get_type (rule)
    if (syntax_rule_is_atomic (rule)) then
        call lex (lexeme, lexer)
        if (debug) write (6, "(A)", advance="no") "Token: "
        if (debug) call lexeme_write (lexeme, 6)
        call parse_node_create_leaf (node, rule, lexeme)
        ok = associated (node)
        if (.not. ok) call lexer_put_back (lexer, lexeme)
    else
        select case (type)
        case (S_ALTERNATIVE); call parse_alternative (node, rule, ok)
        case (S_GROUP);       call parse_group (node, rule, ok)
        case (S_SEQUENCE);    call parse_sequence (node, rule, .false., ok)
        case (S_LIST);        call parse_sequence (node, rule, .true., ok)
        case (S_ARGS);        call parse_args (node, rule, ok)
        case (S_IGNORE);      call parse_ignore (node, ok)
        end select
    end if
    if (debug) then
        if (ok) then
            write (6, "(A)", advance="no") "Matched rule: "
        else

```



```

        write (6, "(A)", advance="no") "Failed rule: "
    end if
    call syntax_rule_write (rule)
    if (associated (node)) call parse_node_write (node)
end if
end subroutine parse_node_match_rule

```

Parse an alternative: try each case. If the match succeeds, the node has been filled, so return. If nothing works, return failure.

```

(Parser: internal subroutines of parse_tree_init)+≡
recursive subroutine parse_alternative (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    integer :: i
    do i = 1, syntax_rule_get_n_sub (rule)
        call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, i), ok)
        if (ok) return
    end do
    ok = .false.
end subroutine parse_alternative

```

Parse a group: the first and third lexemes have to be the delimiters, the second one is parsed as the actual node, using now the child rule. If the first match fails, return with failure. If the other matches fail, issue an error, since we cannot lex back more than one item.

```

(Parser: internal subroutines of parse_tree_init)+≡
recursive subroutine parse_group (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    type(string_t), dimension(2) :: delimiter
    delimiter = syntax_rule_get_delimiter (rule)
    call lex (lexeme, lexer)
    if (lexeme_get_string (lexeme) == delimiter(1)) then
        call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, 1), ok)
        if (ok) then
            call lex (lexeme, lexer)
            if (lexeme_get_string (lexeme) == delimiter(2)) then
                ok = .true.
            else
                call parse_error (rule, lexeme)
            end if
        else
            call parse_error (rule, lexeme)
        end if
    else
        call lexer_put_back (lexer, lexeme)
        ok = .false.
    end if
end subroutine parse_group

```

Parsing a sequence. The last rule element may be special: optional and/or repetitive. Each sub-node that matches is appended to the sub-node list of the parent node.

If `sep` is true, we look for a separator after each element.

```

(Parser: internal subroutines of parse_tree_init)+≡
recursive subroutine parse_sequence (node, rule, sep, ok)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  logical, intent(in) :: sep
  logical, intent(out) :: ok
  type(parse_node_t), pointer :: current
  integer :: i, n
  logical :: opt, rep, cont
  type(string_t) :: separator
  call parse_node_create_branch (node, rule)
  if (sep) separator = syntax_rule_get_separator (rule)
  n = syntax_rule_get_n_sub (rule)
  opt = syntax_rule_last_optional (rule)
  rep = syntax_rule_last_repetitive (rule)
  ok = .true.
  cont = .true.
  SCAN_RULE: do i = 1, n
    call parse_node_match_rule &
      (current, syntax_rule_get_sub_ptr (rule, i), cont)
    if (cont) then
      call parse_node_append_sub (node, current)
      if (sep .and. (i<n .or. rep)) then
        call lex (lexeme, lexer)
        if (lexeme_get_string (lexeme) /= separator) then
          call lexer_put_back (lexer, lexeme)
          cont = .false.
          exit SCAN_RULE
        end if
      end if
    else
      if (i == n .and. opt) then
        exit SCAN_RULE
      else if (i == 1) then
        ok = .false.
        exit SCAN_RULE
      else
        call parse_error (rule, lexeme)
      end if
    end if
  end do SCAN_RULE
  if (rep) then
    do while (cont)
      call parse_node_match_rule &
        (current, syntax_rule_get_sub_ptr (rule, n), cont)
      if (cont) then
        call parse_node_append_sub (node, current)
        if (sep) then
          call lex (lexeme, lexer)
          if (lexeme_get_string (lexeme) /= separator) then
            call lexer_put_back (lexer, lexeme)
            cont = .false.
          end if
        end if
      end if
    end while
  end if
end subroutine parse_sequence

```

```

        end if
    else
        if (sep) call parse_error (rule, lexeme)
        end if
    end do
end if
call parse_node_freeze_branch (node)
end subroutine parse_sequence

```

Argument list: We use the `parse_group` code, but call `parse_sequence` inside.

```

(Parser: internal subroutines of parse_tree_init)+≡
recursive subroutine parse_args (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    type(string_t), dimension(2) :: delimiter
    delimiter = syntax_rule_get_delimiter (rule)
    call lex (lexeme, lexer)
    if (lexeme_get_string (lexeme) == delimiter(1)) then
        call parse_sequence (node, rule, .true., ok)
        if (ok) then
            call lex (lexeme, lexer)
            if (lexeme_get_string (lexeme) == delimiter(2)) then
                ok = .true.
            else
                call parse_error (rule, lexeme)
            end if
        else
            call parse_error (rule, lexeme)
        end if
    else
        call lexer_put_back (lexer, lexeme)
        ok = .false.
    end if
end subroutine parse_args

```

The IGNORE syntax reads one lexeme and discards it if it is numeric, logical or string/identifier (but not a keyword). This is a successful match. Otherwise, the match fails. The node pointer is returned disassociated in any case.

```

(Parser: internal subroutines of parse_tree_init)+≡
subroutine parse_ignore (node, ok)
    type(parse_node_t), pointer :: node
    logical, intent(out) :: ok
    call lex (lexeme, lexer)
    select case (lexeme_get_type (lexeme))
    case (T_NUMERIC, T_IDENTIFIER, T_QUOTED)
        ok = .true.
    case default
        ok = .false.
    end select
    node => null ()
end subroutine parse_ignore

```

If the match fails and we cannot step back:

```

(Parser: internal subroutines of parse_tree_init)+≡

```

```

subroutine parse_error (rule, lexeme)
  type(syntax_rule_t), intent(in) :: rule
  type(lexeme_t), intent(in) :: lexeme
  character(80) :: buffer
  integer :: u, iostat
  call lexer_show_location (lexer)
  u = free_unit ()
  open (u, status = "scratch")
  write (u, "(A)", advance="no") "Expected syntax:"
  call syntax_rule_write (rule, u)
  write (u, "(A)", advance="no") "Found token:"
  call lexeme_write (lexeme, u)
  rewind (u)
  do
    read (u, "(A)", iostat=iostat) buffer
    if (iostat /= 0) exit
    call msg_message (trim (buffer))
  end do
  call msg_fatal (" Syntax error " &
    // "(at or before the location indicated above)")
end subroutine parse_error

```

The finalizer recursively deallocates all nodes and their contents. For each node, `parse_node_final` is called on the sub-nodes, which in turn deallocates the token or sub-node array contained within each of them. At the end, only the top node remains to be deallocated.

```

(Parser: public)+≡
  public :: parse_tree_final

(Parser: procedures)+≡
  subroutine parse_tree_final (parse_tree)
    type(parse_tree_t), intent(inout) :: parse_tree
    if (associated (parse_tree%root_node)) then
      call parse_node_final (parse_tree%root_node)
      deallocate (parse_tree%root_node)
    end if
  end subroutine parse_tree_final

```

Print the parse tree. Print one token per line, indented according to the depth of the node.

The `verbose` version includes type identifiers for the nodes.

```

(Parser: public)+≡
  public :: parse_tree_write

(Parser: procedures)+≡
  subroutine parse_tree_write (parse_tree, unit, verbose)
    type(parse_tree_t), intent(in) :: parse_tree
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    logical :: short
    u = output_unit (unit); if (u < 0) return
    short = .true.; if (present (verbose)) short = .not. verbose
    write (u, "(A)") "Parse tree:"

```

```

    if (associated (parse_tree%root_node)) then
        call parse_node_write_rec (parse_tree%root_node, unit, short, 1)
    else
        write (u, *) "[empty]"
    end if
end subroutine parse_tree_write

```

This is a generic error that can be issued if the parse tree does not meet the expectations of the parser. This most certainly indicates a bug.

```

<Parser: public>+≡
    public :: parse_tree_bug

<Parser: procedures>+≡
    subroutine parse_tree_bug (node, keys)
        type(parse_node_t), intent(in) :: node
        character(*), intent(in) :: keys
        call parse_node_write (node)
        call msg_bug (" Inconsistency in parse tree: expected " // keys)
    end subroutine parse_tree_bug

```

### 3.4.7 Access the parser

For scanning the parse tree we give access to the top node, as a node pointer. Of course, there should be no write access.

```

<Parser: public>+≡
    public :: parse_tree_get_root_ptr

<Parser: procedures>+≡
    function parse_tree_get_root_ptr (parse_tree) result (node)
        type(parse_node_t), pointer :: node
        type(parse_tree_t), intent(in), target :: parse_tree
        node => parse_tree%root_node
    end function parse_tree_get_root_ptr

```

### 3.4.8 Tools

This operation traverses the parse tree and simplifies any occurrences of a set of syntax rules. If such a parse node has only one sub-node, it is replaced by that subnode. (This makes sense only of the rules to eliminate have no meaningful token.)

```

<Parser: public>+≡
    public :: parse_tree_reduce

<Parser: procedures>+≡
    subroutine parse_tree_reduce (parse_tree, rule_key)
        type(parse_tree_t), intent(inout) :: parse_tree
        type(string_t), dimension(:), intent(in) :: rule_key
        type(parse_node_t), pointer :: pn
        pn => parse_tree%root_node
        if (associated (pn)) then
            call parse_node_reduce (pn, null(), null())
        end if
    end subroutine parse_tree_reduce

```

```

end if
contains
recursive subroutine parse_node_reduce (pn, pn_prev, pn_parent)
  type(parse_node_t), intent(inout), pointer :: pn
  type(parse_node_t), intent(in), pointer :: pn_prev, pn_parent
  type(parse_node_t), pointer :: pn_sub, pn_sub_prev, pn_tmp
  pn_sub_prev => null ()
  pn_sub => pn%sub_first
  do while (associated (pn_sub))
    call parse_node_reduce (pn_sub, pn_sub_prev, pn)
    pn_sub_prev => pn_sub
    pn_sub => pn_sub%next
  end do
  if (parse_node_get_n_sub (pn) == 1) then
    if (matches (parse_node_get_rule_key (pn), rule_key)) then
      pn_tmp => pn
      pn => pn%sub_first
      if (associated (pn_prev)) then
        pn_prev%next => pn
      else if (associated (pn_parent)) then
        pn_parent%sub_first => pn
      else
        parse_tree%root_node => pn
      end if
      if (associated (pn_tmp%next)) then
        pn%next => pn_tmp%next
      else if (associated (pn_parent)) then
        pn_parent%sub_last => pn
      end if
      call parse_node_final (pn_tmp, recursive=.false.)
      deallocate (pn_tmp)
    end if
  end if
end subroutine parse_node_reduce
function matches (key, key_list) result (flag)
  logical :: flag
  type(string_t), intent(in) :: key
  type(string_t), dimension(:), intent(in) :: key_list
  integer :: i
  flag = .true.
  do i = 1, size (key_list)
    if (key == key_list(i)) return
  end do
  flag = .false.
end function matches
end subroutine parse_tree_reduce

```

### 3.4.9 Applications

For a file of the form

```

process foo, bar
  <something>

```

```

process xyz
  <something>

```

get the <something> entry node for the first matching process tag. If no matching entry is found, the node pointer remains unassociated.

*<Parser: public>+≡*

```

public :: parse_tree_get_process_ptr

```

*<Parser: procedures>+≡*

```

function parse_tree_get_process_ptr (parse_tree, process) result (node)
  type(parse_node_t), pointer :: node
  type(parse_tree_t), intent(in), target :: parse_tree
  type(string_t), intent(in) :: process
  type(parse_node_t), pointer :: node_root, node_process_def
  type(parse_node_t), pointer :: node_process_phs, node_process_list
  integer :: j
  node_root => parse_tree_get_root_ptr (parse_tree)
  if (associated (node_root)) then
    node_process_phs => parse_node_get_sub_ptr (node_root)
    SCAN_FILE: do while (associated (node_process_phs))
      node_process_def => parse_node_get_sub_ptr (node_process_phs)
      node_process_list => parse_node_get_sub_ptr (node_process_def, 2)
      do j = 1, parse_node_get_n_sub (node_process_list)
        if (parse_node_get_string &
          (parse_node_get_sub_ptr (node_process_list, j)) &
          == process) then
          node => parse_node_get_next_ptr (node_process_def)
          return
        end if
      end do
      node_process_phs => parse_node_get_next_ptr (node_process_phs)
    end do SCAN_FILE
    node => null ()
  else
    node => null ()
  end if
end function parse_tree_get_process_ptr

```

### 3.4.10 Test the parser

*<Parser: public>+≡*

```

public :: parse_test

```

*<Parser: procedures>+≡*

```

subroutine parse_test
  use ifiles
  use lexers

  type(ifile_t) :: ifile
  type(syntax_t), target :: syntax
  type(lexer_t) :: lexer
  type(stream_t), target :: stream
  type(parse_tree_t), target :: parse_tree

```

```

print "(A)", "Parser test"
print *

call ifile_append (ifile, "SEQ expr = term addition*")
call ifile_append (ifile, "SEQ addition = plus_or_minus term")
call ifile_append (ifile, "SEQ term = factor multiplication*")
call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
call ifile_append (ifile, "SEQ factor = atom exponentiation*")
call ifile_append (ifile, "SEQ exponentiation = '^' atom")
call ifile_append (ifile, "ALT atom = real | delimited_expr")
call ifile_append (ifile, "GRO delimited_expr = ( expr )")
call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
call ifile_append (ifile, "KEY '+'")
call ifile_append (ifile, "KEY '-'")
call ifile_append (ifile, "KEY '*'")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '^'")
call ifile_append (ifile, "REA real")

print "(A)", "File contents (syntax definition):"
call ifile_write (ifile)
print "(A)", "EOF"
print *

call syntax_init (syntax, ifile)
call ifile_final (ifile)
call syntax_write (syntax)
print *

call lexer_init (lexer, &
    comment_chars = "", &
    quote_chars = "'", &
    quote_match = "'", &
    single_chars = "+-*/^()", &
    special_class = (/ "" /), &
    keyword_list = syntax_get_keyword_list_ptr (syntax))
call lexer_write_setup (lexer)
print *

call ifile_append (ifile, "(27+8^3-2/3)*(4+7)^2*99")
print "(A)", "File contents (input file):"
call ifile_write (ifile)
print "(A)", "EOF"
print *

call stream_init (stream, ifile)
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax, lexer)
call stream_final (stream)
call parse_tree_write (parse_tree)
print *

print "(A)", "Cleanup, everything should now be empty:"

```



```

print *

call parse_tree_final (parse_tree)
call parse_tree_write (parse_tree)
print *

call lexer_final (lexer)
call lexer_write_setup (lexer)
print *

call ifile_final (ifile)
print "(A)", "File contents:"
call ifile_write (ifile)
print "(A)", "EOF"
print *

call syntax_final (syntax)
call syntax_write (syntax)

end subroutine parse_test

```

## Chapter 4

# Physics library

This part consists of two modules:

**constants** Physical and mathematical parameters that never change. The file has been moved to the **src/misc** subdirectory of the **WHIZARD** project.

**c\_particles** A simple data type for particles which is C compatible.

**lorentz** Define three-vectors, four-vectors and Lorentz transformations and common operations for them.

**sm\_physics** Here, running functions are stored for special kinematical setup like running coupling constants, Catani-Seymour dipoles, or Sudakov factors.

## 4.1 C-compatible Particle Type

For easy communication with C code, we introduce a simple C-compatible type for particles. It has the contents of the `prt_t` type defined in the `subevents` module below (except the source information).

The `c_prt` type is transparent, and its contents should be regarded as part of the interface.

```

⟨c_particles.f90⟩≡
  ⟨File header⟩

  module c_particles

    use iso_c_binding !NODEP!
    ⟨Use file utils⟩

    ⟨Standard module head⟩

    ⟨C Particles: public⟩

    ⟨C Particles: types⟩

    contains

    ⟨C Particles: procedures⟩
  end module c_particles

⟨C Particles: public⟩≡
  public :: c_prt_t

⟨C Particles: types⟩≡
  type, bind(C) :: c_prt_t
    integer(c_int) :: type = 0
    integer(c_int) :: pdg = 0
    integer(c_int) :: polarized = 0
    integer(c_int) :: h = 0
    real(c_double) :: pe = 0
    real(c_double) :: px = 0
    real(c_double) :: py = 0
    real(c_double) :: pz = 0
    real(c_double) :: p2 = 0
  end type c_prt_t

```

This is for debugging only, there is no C binding. It is a simplified version of `prt_write`.

```

⟨C Particles: public⟩+≡
  public :: c_prt_write

⟨C Particles: procedures⟩≡
  subroutine c_prt_write (prt, unit)
    type(c_prt_t), intent(in) :: prt
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A)", advance="no") "prt("

```

```

write (u, "(I0,':')", advance="no") prt%type
if (prt%polarized /= 0) then
  write (u, "(I0,'/',I0,'|')", advance="no") prt%pdg, prt%h
else
  write (u, "(I0,'|')", advance="no") prt%pdg
end if
write (u, "(ES14.7,';',ES14.7,';',ES14.7,';',ES14.7)", advance="no") &
  prt%pe, prt%px, prt%py, prt%pz
write (u, "('|',ES19.12)", advance="no") prt%p2
write (u, "(A)" " ")
end subroutine c_prt_write

```

## 4.2 Lorentz algebra

Define Lorentz vectors, three-vectors, boosts, and some functions to manipulate them.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

```
<lorentz.f90>≡  
  <File header>  
  
  module lorentz  
  
    <Use kinds>  
    use constants, only: pi, twopi, degree !NODEP!  
    <Use file utils>  
    use diagnostics !NODEP!  
    use c_particles  
  
    <Standard module head>  
  
    <Lorentz: public>  
  
    <Lorentz: public operators>  
  
    <Lorentz: public functions>  
  
    <Lorentz: types>  
  
    <Lorentz: parameters>  
  
    <Lorentz: interfaces>  
  
    contains  
  
    <Lorentz: procedures>  
  end module lorentz
```

### 4.2.1 Three-vectors

First of all, let us introduce three-vectors in a trivial way. The functions and overloaded elementary operations clearly are too much overhead, but we like to keep the interface for three-vectors and four-vectors exactly parallel. By the way, we might attach a label to a vector by extending the type definition later.

```
<Lorentz: public>≡  
  public :: vector3_t  
  
<Lorentz: types>≡  
  type :: vector3_t  
    private  
    real(default), dimension(3) :: p  
  end type vector3_t
```

Output a vector

```

(Lorentz: public)+≡
    public :: vector3_write

(Lorentz: procedures)≡
    subroutine vector3_write (p, unit)
        type(vector3_t), intent(in) :: p
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write(u, 1) 'P = ', p%p
    1    format (1x,A,3(1x,ES19.12))
    end subroutine vector3_write

```

This is a three-vector with zero components

```

(Lorentz: public)+≡
    public :: vector3_null

(Lorentz: parameters)≡
    type(vector3_t), parameter :: vector3_null = &
        vector3_t ((/ 0._default, 0._default, 0._default /))

```

Canonical three-vector:

```

(Lorentz: public)+≡
    public :: vector3_canonical

(Lorentz: procedures)+≡
    elemental function vector3_canonical (k) result (p)
        type(vector3_t) :: p
        integer, intent(in) :: k
        p = vector3_null
        p%p(k) = 1
    end function vector3_canonical

```

A moving particle ( $k$ -axis, or arbitrary axis). Note that the function for the generic momentum cannot be elemental.

```

(Lorentz: public)+≡
    public :: vector3_moving

(Lorentz: interfaces)≡
    interface vector3_moving
        module procedure vector3_moving_canonical
        module procedure vector3_moving_generic
    end interface

(Lorentz: procedures)+≡
    elemental function vector3_moving_canonical (p, k) result(q)
        type(vector3_t) :: q
        real(default), intent(in) :: p
        integer, intent(in) :: k
        q = vector3_null
        q%p(k) = p
    end function vector3_moving_canonical
    pure function vector3_moving_generic (p) result(q)

```

```

    real(default), dimension(3), intent(in) :: p
    type(vector3_t) :: q
    q%p = p
end function vector3_moving_generic

```

Equality and inequality

```

<Lorentz: public operators>≡
    public :: operator(==), operator(/=)

<Lorentz: interfaces>+≡
    interface operator(==)
        module procedure vector3_eq
    end interface
    interface operator(/=)
        module procedure vector3_neq
    end interface

<Lorentz: procedures>+≡
    elemental function vector3_eq (p, q) result (r)
        logical :: r
        type(vector3_t), intent(in) :: p,q
        r = all (p%p == q%p)
    end function vector3_eq
    elemental function vector3_neq (p, q) result (r)
        logical :: r
        type(vector3_t), intent(in) :: p,q
        r = any (p%p /= q%p)
    end function vector3_neq

```

Define addition and subtraction

```

<Lorentz: public operators>+≡
    public :: operator(+), operator(-)

<Lorentz: interfaces>+≡
    interface operator(+)
        module procedure add_vector3
    end interface
    interface operator(-)
        module procedure sub_vector3
    end interface

<Lorentz: procedures>+≡
    elemental function add_vector3 (p, q) result (r)
        type(vector3_t) :: r
        type(vector3_t), intent(in) :: p,q
        r%p = p%p + q%p
    end function add_vector3
    elemental function sub_vector3 (p, q) result (r)
        type(vector3_t) :: r
        type(vector3_t), intent(in) :: p,q
        r%p = p%p - q%p
    end function sub_vector3

```

The multiplication sign is overloaded with scalar multiplication; similarly division:

```

(Lorentz: public operators)+≡
    public :: operator(*), operator(/)

(Lorentz: interfaces)+≡
    interface operator(*)
        module procedure prod_integer_vector3, prod_vector3_integer
        module procedure prod_real_vector3, prod_vector3_real
    end interface
    interface operator(/)
        module procedure div_vector3_real, div_vector3_integer
    end interface

(Lorentz: procedures)+≡
    elemental function prod_real_vector3 (s, p) result (q)
        type(vector3_t) :: q
        real(default), intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = s * p%p
    end function prod_real_vector3
    elemental function prod_vector3_real (p, s) result (q)
        type(vector3_t) :: q
        real(default), intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = s * p%p
    end function prod_vector3_real
    elemental function div_vector3_real (p, s) result (q)
        type(vector3_t) :: q
        real(default), intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = p%p/s
    end function div_vector3_real
    elemental function prod_integer_vector3 (s, p) result (q)
        type(vector3_t) :: q
        integer, intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = s * p%p
    end function prod_integer_vector3
    elemental function prod_vector3_integer (p, s) result (q)
        type(vector3_t) :: q
        integer, intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = s * p%p
    end function prod_vector3_integer
    elemental function div_vector3_integer (p, s) result (q)
        type(vector3_t) :: q
        integer, intent(in) :: s
        type(vector3_t), intent(in) :: p
        q%p = p%p/s
    end function div_vector3_integer

```

The multiplication sign can also indicate scalar products:

```

(Lorentz: interfaces)+≡

```



```

interface operator(*)
  module procedure prod_vector3
end interface

<Lorentz: procedures>+≡
  elemental function prod_vector3 (p, q) result (s)
    real(default) :: s
    type(vector3_t), intent(in) :: p,q
    s = dot_product (p%p, q%p)
  end function prod_vector3

<Lorentz: public functions>≡
  public :: cross_product

<Lorentz: interfaces>+≡
  interface cross_product
    module procedure vector3_cross_product
  end interface

<Lorentz: procedures>+≡
  elemental function vector3_cross_product (p, q) result (r)
    type(vector3_t) :: r
    type(vector3_t), intent(in) :: p,q
    integer :: i
    do i=1,3
      r%p(i) = dot_product (p%p, matmul(epsilon_three(i,:,:), q%p))
    end do
  end function vector3_cross_product

```

Exponentiation is defined only for integer powers. Odd powers mean take the square root; so `p**1` is the length of `p`.

```

<Lorentz: public operators>+≡
  public :: operator(**)

<Lorentz: interfaces>+≡
  interface operator(**)
    module procedure power_vector3
  end interface

<Lorentz: procedures>+≡
  elemental function power_vector3 (p, e) result (s)
    real(default) :: s
    type(vector3_t), intent(in) :: p
    integer, intent(in) :: e
    s = dot_product (p%p, p%p)
    if (e/=2) then
      if (mod(e,2)==0) then
        s = s**(e/2)
      else
        s = sqrt(s)**e
      end if
    end if
  end function power_vector3

```

Finally, we need a negation.

```

<Lorentz: interfaces>+≡
  interface operator(-)
    module procedure negate_vector3
  end interface

<Lorentz: procedures>+≡
  elemental function negate_vector3 (p) result (q)
    type(vector3_t) :: q
    type(vector3_t), intent(in) :: p
    integer :: i
    do i = 1, 3
      if (p%p(i) == -p%p(i)) then
        q%p(i) = 0
      else
        q%p(i) = -p%p(i)
      end if
    end do
  end function negate_vector3

```

The sum function can be useful:

```

<Lorentz: public functions>+≡
  public :: sum

<Lorentz: interfaces>+≡
  interface sum
    module procedure sum_vector3
  end interface

```

There used to be a mask here, but the Intel compiler crashes with it

```

<Lorentz: procedures>+≡
  pure function sum_vector3 (p) result (q)
    type(vector3_t) :: q
    type(vector3_t), dimension(:), intent(in) :: p
    integer :: i
    do i=1, 3
      q%p(i) = sum (p%p(i))
    end do
  end function sum_vector3
!  pure function sum_vector3_mask (p, mask) result (q)
!    type(vector3_t) :: q
!    type(vector3_t), dimension(:), intent(in) :: p
!    logical, dimension(:), intent(in) :: mask
!    integer :: i
!    do i=1, 3
!      q%p(i) = sum (p%p(i), mask=mask)
!    end do
!  end function sum_vector3_mask

```

Any component:

```

<Lorentz: public>+≡
  public :: vector3_get_component

```

```

⟨Lorentz: procedures⟩+≡
  elemental function vector3_get_component (p, k) result (c)
    type(vector3_t), intent(in) :: p
    integer, intent(in) :: k
    real(default) :: c
    c = p%p(k)
  end function vector3_get_component

```

Extract all components. This is not elemental.

```

⟨Lorentz: public⟩+≡
  public :: vector3_get_components

⟨Lorentz: procedures⟩+≡
  pure function vector3_get_components (p) result (a)
    type(vector3_t), intent(in) :: p
    real(default), dimension(3) :: a
    a = p%p
  end function vector3_get_components

```

This function returns the direction of a three-vector, i.e., a normalized three-vector. If the vector is null, we return a null vector.

```

⟨Lorentz: public functions⟩+≡
  public :: direction

⟨Lorentz: interfaces⟩+≡
  interface direction
    module procedure vector3_get_direction
  end interface

⟨Lorentz: procedures⟩+≡
  elemental function vector3_get_direction (p) result (q)
    type(vector3_t) :: q
    type(vector3_t), intent(in) :: p
    real(default) :: pp
    pp = p**1
    if (pp /= 0) then
      q%p = p%p / pp
    else
      q%p = 0
    end if
  end function vector3_get_direction

```

#### 4.2.2 Four-vectors

In four-vectors the zero-component needs special treatment, therefore we do not use the standard operations. Sure, we pay for the extra layer of abstraction by losing efficiency; so we have to assume that the time-critical applications do not involve four-vector operations.

```

⟨Lorentz: public⟩+≡
  public :: vector4_t

```

```

<Lorentz: types>+≡
  type :: vector4_t
  private
    real(default), dimension(0:3) :: p = &
      (/ 0._default, 0._default, 0._default, 0._default /)
  end type vector4_t

```

Output a vector

```

<Lorentz: public>+≡
  public :: vector4_write

<Lorentz: procedures>+≡
  subroutine vector4_write (p, unit, show_mass)
    type(vector4_t), intent(in) :: p
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: show_mass
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write(u, 1) 'E = ', p%p(0)
    write(u, 1) 'P = ', p%p(1:)
    if (present (show_mass)) then
      if (show_mass) &
        write (u, 1) 'M = ', p**1
    end if
1   format (1x,A,3(1x,ES19.12))
  end subroutine vector4_write

```

Binary I/O

```

<Lorentz: public>+≡
  public :: vector4_write_raw
  public :: vector4_read_raw

<Lorentz: procedures>+≡
  subroutine vector4_write_raw (p, u)
    type(vector4_t), intent(in) :: p
    integer, intent(in) :: u
    write (u) p%p
  end subroutine vector4_write_raw

  subroutine vector4_read_raw (p, u, iostat)
    type(vector4_t), intent(out) :: p
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    read (u, iostat=iostat) p%p
  end subroutine vector4_read_raw

```

This is a four-vector with zero components

```

<Lorentz: public>+≡
  public :: vector4_null

<Lorentz: parameters>+≡
  type(vector4_t), parameter :: vector4_null = &
    vector4_t (/ 0._default, 0._default, 0._default, 0._default /)

```

Canonical four-vector:

```

(Lorentz: public)+≡
    public :: vector4_canonical

(Lorentz: procedures)+≡
    elemental function vector4_canonical (k) result (p)
        type(vector4_t) :: p
        integer, intent(in) :: k
        p = vector4_null
        p%k = 1
    end function vector4_canonical

```

A particle at rest:

```

(Lorentz: public)+≡
    public :: vector4_at_rest

(Lorentz: procedures)+≡
    elemental function vector4_at_rest (m) result (p)
        type(vector4_t) :: p
        real(default), intent(in) :: m
        p = vector4_t (/ m, 0._default, 0._default, 0._default /)
    end function vector4_at_rest

```

A moving particle ( $k$ -axis, or arbitrary axis)

```

(Lorentz: public)+≡
    public :: vector4_moving

(Lorentz: interfaces)+≡
    interface vector4_moving
        module procedure vector4_moving_canonical
        module procedure vector4_moving_generic
    end interface

(Lorentz: procedures)+≡
    elemental function vector4_moving_canonical (E, p, k) result (q)
        type(vector4_t) :: q
        real(default), intent(in) :: E, p
        integer, intent(in) :: k
        q = vector4_at_rest(E)
        q%p(k) = p
    end function vector4_moving_canonical
    elemental function vector4_moving_generic (E, p) result (q)
        type(vector4_t) :: q
        real(default), intent(in) :: E
        type(vector3_t), intent(in) :: p
        q%p(0) = E
        q%p(1:) = p%p
    end function vector4_moving_generic

```

Equality and inequality

```

(Lorentz: interfaces)+≡
    interface operator(==)
        module procedure vector4_eq
    end interface

```

```

interface operator(/=)
  module procedure vector4_neq
end interface

<Lorentz: procedures>+≡
elemental function vector4_eq (p, q) result (r)
  logical :: r
  type(vector4_t), intent(in) :: p,q
  r = all (p%p == q%p)
end function vector4_eq
elemental function vector4_neq (p, q) result (r)
  logical :: r
  type(vector4_t), intent(in) :: p,q
  r = any (p%p /= q%p)
end function vector4_neq

```

Addition and subtraction:

```

<Lorentz: interfaces>+≡
interface operator(+)
  module procedure add_vector4
end interface
interface operator(-)
  module procedure sub_vector4
end interface

<Lorentz: procedures>+≡
elemental function add_vector4 (p,q) result (r)
  type(vector4_t) :: r
  type(vector4_t), intent(in) :: p,q
  r%p = p%p + q%p
end function add_vector4
elemental function sub_vector4 (p,q) result (r)
  type(vector4_t) :: r
  type(vector4_t), intent(in) :: p,q
  r%p = p%p - q%p
end function sub_vector4

```

We also need scalar multiplication and division:

```

<Lorentz: interfaces>+≡
interface operator(*)
  module procedure prod_real_vector4, prod_vector4_real
  module procedure prod_integer_vector4, prod_vector4_integer
end interface
interface operator(/)
  module procedure div_vector4_real
  module procedure div_vector4_integer
end interface

<Lorentz: procedures>+≡
elemental function prod_real_vector4 (s, p) result (q)
  type(vector4_t) :: q
  real(default), intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = s * p%p

```

```

end function prod_real_vector4
elemental function prod_vector4_real (p, s) result (q)
  type(vector4_t) :: q
  real(default), intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector4_real
elemental function div_vector4_real (p, s) result (q)
  type(vector4_t) :: q
  real(default), intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = p%p/s
end function div_vector4_real
elemental function prod_integer_vector4 (s, p) result (q)
  type(vector4_t) :: q
  integer, intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = s * p%p
end function prod_integer_vector4
elemental function prod_vector4_integer (p, s) result (q)
  type(vector4_t) :: q
  integer, intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector4_integer
elemental function div_vector4_integer (p, s) result (q)
  type(vector4_t) :: q
  integer, intent(in) :: s
  type(vector4_t), intent(in) :: p
  q%p = p%p/s
end function div_vector4_integer

```

Scalar products and squares in the Minkowski sense:

```

<Lorentz: interfaces>+≡
  interface operator(*)
    module procedure prod_vector4
  end interface
  interface operator(**)
    module procedure power_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function prod_vector4 (p, q) result (s)
    real(default) :: s
    type(vector4_t), intent(in) :: p,q
    s = p%p(0)*q%p(0) - dot_product(p%p(1:), q%p(1:))
  end function prod_vector4

```

The power operation for four-vectors is signed, i.e.,  $p^{**1}$  is positive for timelike and negative for spacelike vectors. Note that  $(p^{**1})^{**2}$  is not necessarily equal to  $p^{**2}$ .

```

<Lorentz: procedures>+≡
  elemental function power_vector4 (p, e) result (s)

```

```

real(default) :: s
type(vector4_t), intent(in) :: p
integer, intent(in) :: e
s = p*p
if (e/=2) then
  if (mod(e,2)==0) then
    s = s**(e/2)
  elseif (s>=0) then
    s = sqrt(s)**e
  else
    s = -(sqrt(abs(s))**e)
  end if
end if
end function power_vector4

```

Finally, we introduce a negation

```

<Lorentz: interfaces>+≡
  interface operator(-)
    module procedure negate_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function negate_vector4 (p) result (q)
    type(vector4_t) :: q
    type(vector4_t), intent(in) :: p
    integer :: i
    do i = 0, 3
      if (p%p(i) == -p%p(i)) then
        q%p(i) = 0
      else
        q%p(i) = -p%p(i)
      end if
    end do
  end function negate_vector4

```

The sum function can be useful:

```

<Lorentz: interfaces>+≡
  interface sum
    module procedure sum_vector4
  end interface

```

There used to be a mask here, but the Intel compiler crashes with it

```

<Lorentz: procedures>+≡
  pure function sum_vector4 (p) result (q)
    type(vector4_t) :: q
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i
    do i=0, 3
      q%p(i) = sum (p%p(i))
    end do
  end function sum_vector4
!   pure function sum_vector4_mask (p, mask) result (q)
!     type(vector4_t) :: q
!     type(vector4_t), dimension(:), intent(in) :: p

```



```

!      logical, dimension(:), intent(in) :: mask
!      integer :: i
!      do i=0, 3
!          q%p(i) = sum (p%p(i), mask=mask)
!      end do
!      end function sum_vector4_mask

```

### 4.2.3 Conversions

Manually set a component of the four-vector:

```

<Lorentz: public>+≡
    public :: vector4_set_component

<Lorentz: procedures>+≡
    subroutine vector4_set_component (p, k, c)
        type(vector4_t), intent(inout) :: p
        integer, intent(in) :: k
        real(default), intent(in) :: c
        p%p(k) = c
    end subroutine vector4_set_component

```

Any component:

```

<Lorentz: public>+≡
    public :: vector4_get_component

<Lorentz: procedures>+≡
    elemental function vector4_get_component (p, k) result (c)
        real(default) :: c
        type(vector4_t), intent(in) :: p
        integer, intent(in) :: k
        c = p%p(k)
    end function vector4_get_component

```

Extract all components. This is not elemental.

```

<Lorentz: public>+≡
    public :: vector4_get_components

<Lorentz: procedures>+≡
    pure function vector4_get_components (p) result (a)
        real(default), dimension(0:3) :: a
        type(vector4_t), intent(in) :: p
        a = p%p
    end function vector4_get_components

```

This function returns the space part of a four-vector, such that we can apply three-vector operations on it:

```

<Lorentz: public functions>+≡
    public :: space_part

<Lorentz: interfaces>+≡
    interface space_part
        module procedure vector4_get_space_part
    end interface

```

```

(Lorentz: procedures)+≡
  elemental function vector4_get_space_part (p) result (q)
    type(vector3_t) :: q
    type(vector4_t), intent(in) :: p
    q%p = p%p(1:)
  end function vector4_get_space_part

```

This function returns the direction of a four-vector, i.e., a normalized three-vector. If the four-vector has zero space part, we return a null vector.

```

(Lorentz: interfaces)+≡
  interface direction
    module procedure vector4_get_direction
  end interface

(Lorentz: procedures)+≡
  elemental function vector4_get_direction (p) result (q)
    type(vector3_t) :: q
    type(vector4_t), intent(in) :: p
    real(default) :: qq
    q%p = p%p(1:)
    qq = q**1
    if (qq /= 0) then
      q%p = q%p / qq
    else
      q%p = 0
    end if
  end function vector4_get_direction

```

This function returns the four-vector as an ordinary array. A second version for an array of four-vectors.

```

(Lorentz: public functions)+≡
  public :: array_from_vector4

(Lorentz: interfaces)+≡
  interface array_from_vector4
    module procedure array_from_vector4_1
    module procedure array_from_vector4_2
  end interface

(Lorentz: procedures)+≡
  pure function array_from_vector4_1 (p) result (a)
    type(vector4_t), intent(in) :: p
    real(default), dimension(0:3) :: a
    a = p%p
  end function array_from_vector4_1

  pure function array_from_vector4_2 (p) result (a)
    type(vector4_t), dimension(:), intent(in) :: p
    real(default), dimension(0:3, size(p)) :: a
    integer :: i
    forall (i=1:size(p))
      a(0:3,i) = p(i)%p
    end forall
  end function array_from_vector4_2

```

Transform the momentum of a `c_prt` object into a four-vector:

```

(Lorentz: public)+≡
    public :: vector4_from_c_prt

(Lorentz: procedures)+≡
    elemental function vector4_from_c_prt (c_prt) result (p)
        type(vector4_t) :: p
        type(c_prt_t), intent(in) :: c_prt
        p%p(0) = c_prt%pe
        p%p(1) = c_prt%px
        p%p(2) = c_prt%py
        p%p(3) = c_prt%pz
    end function vector4_from_c_prt

```

Initialize a `c_prt_t` object with the components of a four-vector as its kinematical entries. Compute the invariant mass, or use the optional mass-squared value instead.

```

(Lorentz: public)+≡
    public :: vector4_to_c_prt

(Lorentz: procedures)+≡
    elemental function vector4_to_c_prt (p, p2) result (c_prt)
        type(c_prt_t) :: c_prt
        type(vector4_t), intent(in) :: p
        real(default), intent(in), optional :: p2
        c_prt%pe = p%p(0)
        c_prt%px = p%p(1)
        c_prt%py = p%p(2)
        c_prt%pz = p%p(3)
        if (present (p2)) then
            c_prt%p2 = p2
        else
            c_prt%p2 = p ** 2
        end if
    end function vector4_to_c_prt

```

#### 4.2.4 Angles

Return the angles in a canonical system. The angle  $\phi$  is defined between  $0 \leq \phi < 2\pi$ . In degenerate cases, return zero.

```

(Lorentz: public functions)+≡
    public :: azimuthal_angle

(Lorentz: interfaces)+≡
    interface azimuthal_angle
        module procedure vector3_azimuthal_angle
        module procedure vector4_azimuthal_angle
    end interface

(Lorentz: procedures)+≡
    elemental function vector3_azimuthal_angle (p) result (phi)
        real(default) :: phi
        type(vector3_t), intent(in) :: p

```

```

    if (any(p%p(1:2)/=0)) then
        phi = atan2(p%p(2), p%p(1))
        if (phi < 0) phi = phi + twopi
    else
        phi = 0
    end if
end function vector3_azimuthal_angle
elemental function vector4_azimuthal_angle (p) result (phi)
    real(default) :: phi
    type(vector4_t), intent(in) :: p
    phi = vector3_azimuthal_angle (space_part (p))
end function vector4_azimuthal_angle

```

Azimuthal angle in degrees

```

<Lorentz: public functions>+≡
    public :: azimuthal_angle_deg

<Lorentz: interfaces>+≡
    interface azimuthal_angle_deg
        module procedure vector3_azimuthal_angle_deg
        module procedure vector4_azimuthal_angle_deg
    end interface

<Lorentz: procedures>+≡
    elemental function vector3_azimuthal_angle_deg (p) result (phi)
        real(default) :: phi
        type(vector3_t), intent(in) :: p
        phi = vector3_azimuthal_angle (p) / degree
    end function vector3_azimuthal_angle_deg
    elemental function vector4_azimuthal_angle_deg (p) result (phi)
        real(default) :: phi
        type(vector4_t), intent(in) :: p
        phi = vector4_azimuthal_angle (p) / degree
    end function vector4_azimuthal_angle_deg

```

The azimuthal distance of two vectors. This is the difference of the azimuthal angles, but cannot be larger than  $\pi$ : The result is between  $-\pi < \Delta\phi \leq \pi$ .

```

<Lorentz: public functions>+≡
    public :: azimuthal_distance

<Lorentz: interfaces>+≡
    interface azimuthal_distance
        module procedure vector3_azimuthal_distance
        module procedure vector4_azimuthal_distance
    end interface

<Lorentz: procedures>+≡
    elemental function vector3_azimuthal_distance (p, q) result (dphi)
        real(default) :: dphi
        type(vector3_t), intent(in) :: p,q
        dphi = vector3_azimuthal_angle (q) - vector3_azimuthal_angle (p)
        if (dphi <= -pi) then
            dphi = dphi + twopi
        else if (dphi > pi) then
            dphi = dphi - twopi
        end if
    end function vector3_azimuthal_distance

```

```

    end if
end function vector3_azimuthal_distance
elemental function vector4_azimuthal_distance (p, q) result (dphi)
    real(default) :: dphi
    type(vector4_t), intent(in) :: p,q
    dphi = vector3_azimuthal_distance &
        (space_part (p), space_part (q))
end function vector4_azimuthal_distance

```

The same in degrees:

```

<Lorentz: public functions>+≡
    public :: azimuthal_distance_deg

<Lorentz: interfaces>+≡
    interface azimuthal_distance_deg
        module procedure vector3_azimuthal_distance_deg
        module procedure vector4_azimuthal_distance_deg
    end interface

<Lorentz: procedures>+≡
    elemental function vector3_azimuthal_distance_deg (p, q) result (dphi)
        real(default) :: dphi
        type(vector3_t), intent(in) :: p,q
        dphi = vector3_azimuthal_distance (p, q) / degree
    end function vector3_azimuthal_distance_deg
    elemental function vector4_azimuthal_distance_deg (p, q) result (dphi)
        real(default) :: dphi
        type(vector4_t), intent(in) :: p,q
        dphi = vector4_azimuthal_distance (p, q) / degree
    end function vector4_azimuthal_distance_deg

```

The polar angle is defined  $0 \leq \theta \leq \pi$ . Note that ATAN2 has the reversed order of arguments: ATAN2(Y,X). Here,  $x$  is the 3-component while  $y$  is the transverse momentum which is always nonnegative. Therefore, the result is nonnegative as well.

```

<Lorentz: public functions>+≡
    public :: polar_angle

<Lorentz: interfaces>+≡
    interface polar_angle
        module procedure polar_angle_vector3
        module procedure polar_angle_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function polar_angle_vector3 (p) result (theta)
        real(default) :: theta
        type(vector3_t), intent(in) :: p
        if (any(p%p/=0)) then
            theta = atan2 (sqrt(p%p(1)**2 + p%p(2)**2), p%p(3))
        else
            theta = 0
        end if
    end function polar_angle_vector3
    elemental function polar_angle_vector4 (p) result (theta)

```

```

    real(default) :: theta
    type(vector4_t), intent(in) :: p
    theta = polar_angle (space_part (p))
end function polar_angle_vector4

```

This is the cosine of the polar angle:  $-1 \leq \cos \theta \leq 1$ .

```

<Lorentz: public functions>+≡
    public :: polar_angle_ct

<Lorentz: interfaces>+≡
    interface polar_angle_ct
        module procedure polar_angle_ct_vector3
        module procedure polar_angle_ct_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function polar_angle_ct_vector3 (p) result (ct)
        real(default) :: ct
        type(vector3_t), intent(in) :: p
        if (any(p%p/=0)) then
            ct = p%p(3) / p**1
        else
            ct = 1
        end if
    end function polar_angle_ct_vector3
    elemental function polar_angle_ct_vector4 (p) result (ct)
        real(default) :: ct
        type(vector4_t), intent(in) :: p
        ct = polar_angle_ct (space_part (p))
    end function polar_angle_ct_vector4

```

The polar angle in degrees.

```

<Lorentz: public functions>+≡
    public :: polar_angle_deg

<Lorentz: interfaces>+≡
    interface polar_angle_deg
        module procedure polar_angle_deg_vector3
        module procedure polar_angle_deg_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function polar_angle_deg_vector3 (p) result (theta)
        real(default) :: theta
        type(vector3_t), intent(in) :: p
        theta = polar_angle (p) / degree
    end function polar_angle_deg_vector3
    elemental function polar_angle_deg_vector4 (p) result (theta)
        real(default) :: theta
        type(vector4_t), intent(in) :: p
        theta = polar_angle (p) / degree
    end function polar_angle_deg_vector4

```

This is the angle enclosed between two three-momenta. If one of the momenta is zero, we return an angle of zero. The range of the result is  $0 \leq \theta \leq \pi$ . If there is only one argument, take the positive  $z$  axis as reference.

```

(Lorentz: public functions)+≡
    public :: enclosed_angle

(Lorentz: interfaces)+≡
    interface enclosed_angle
        module procedure enclosed_angle_vector3
        module procedure enclosed_angle_vector4
    end interface

(Lorentz: procedures)+≡
    elemental function enclosed_angle_vector3 (p, q) result (theta)
        real(default) :: theta
        type(vector3_t), intent(in) :: p, q
        theta = acos (enclosed_angle_ct (p, q))
    end function enclosed_angle_vector3
    elemental function enclosed_angle_vector4 (p, q) result (theta)
        real(default) :: theta
        type(vector4_t), intent(in) :: p, q
        theta = enclosed_angle (space_part (p), space_part (q))
    end function enclosed_angle_vector4

```

The cosine of the enclosed angle.

```

(Lorentz: public functions)+≡
    public :: enclosed_angle_ct

(Lorentz: interfaces)+≡
    interface enclosed_angle_ct
        module procedure enclosed_angle_ct_vector3
        module procedure enclosed_angle_ct_vector4
    end interface

(Lorentz: procedures)+≡
    elemental function enclosed_angle_ct_vector3 (p, q) result (ct)
        real(default) :: ct
        type(vector3_t), intent(in) :: p, q
        if (any(p%p/=0).and.any(q%p/=0)) then
            ct = p*q / (p**1 * q**1)
            if (ct>1) then
                ct = 1
            else if (ct<-1) then
                ct = -1
            end if
        else
            ct = 1
        end if
    end function enclosed_angle_ct_vector3
    elemental function enclosed_angle_ct_vector4 (p, q) result (ct)
        real(default) :: ct
        type(vector4_t), intent(in) :: p, q
        ct = enclosed_angle_ct (space_part (p), space_part (q))
    end function enclosed_angle_ct_vector4

```

The enclosed angle in degrees.

```

(Lorentz: public functions)+≡
    public :: enclosed_angle_deg

(Lorentz: interfaces)+≡
    interface enclosed_angle_deg
        module procedure enclosed_angle_deg_vector3
        module procedure enclosed_angle_deg_vector4
    end interface

(Lorentz: procedures)+≡
    elemental function enclosed_angle_deg_vector3 (p, q) result (theta)
        real(default) :: theta
        type(vector3_t), intent(in) :: p, q
        theta = enclosed_angle (p, q) / degree
    end function enclosed_angle_deg_vector3
    elemental function enclosed_angle_deg_vector4 (p, q) result (theta)
        real(default) :: theta
        type(vector4_t), intent(in) :: p, q
        theta = enclosed_angle (p, q) / degree
    end function enclosed_angle_deg_vector4

```

The polar angle of the first momentum w.r.t. the second momentum, evaluated in the rest frame of the second momentum. If the second four-momentum is not timelike, return zero.

```

(Lorentz: public functions)+≡
    public :: enclosed_angle_rest_frame
    public :: enclosed_angle_ct_rest_frame
    public :: enclosed_angle_deg_rest_frame

(Lorentz: interfaces)+≡
    interface enclosed_angle_rest_frame
        module procedure enclosed_angle_rest_frame_vector4
    end interface
    interface enclosed_angle_ct_rest_frame
        module procedure enclosed_angle_ct_rest_frame_vector4
    end interface
    interface enclosed_angle_deg_rest_frame
        module procedure enclosed_angle_deg_rest_frame_vector4
    end interface

(Lorentz: procedures)+≡
    elemental function enclosed_angle_rest_frame_vector4 (p, q) result (theta)
        type(vector4_t), intent(in) :: p, q
        real(default) :: theta
        theta = acos (enclosed_angle_ct_rest_frame (p, q))
    end function enclosed_angle_rest_frame_vector4
    elemental function enclosed_angle_ct_rest_frame_vector4 (p, q) result (ct)
        type(vector4_t), intent(in) :: p, q
        real(default) :: ct
        if (invariant_mass(q) > 0) then
            ct = enclosed_angle_ct ( &
                space_part (boost(-q, invariant_mass (q)) * p), &
                space_part (q))
        else

```



```

        ct = 1
    end if
end function enclosed_angle_ct_rest_frame_vector4
elemental function enclosed_angle_deg_rest_frame_vector4 (p, q) &
    result (theta)
    type(vector4_t), intent(in) :: p, q
    real(default) :: theta
    theta = enclosed_angle_rest_frame (p, q) / degree
end function enclosed_angle_deg_rest_frame_vector4

```

## 4.2.5 More kinematical functions (some redundant)

The scalar transverse momentum (assuming the  $z$  axis is longitudinal)

```

<Lorentz: public functions>+≡
    public :: transverse_part

<Lorentz: interfaces>+≡
    interface transverse_part
        module procedure transverse_part_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function transverse_part_vector4 (p) result (pT)
        real(default) :: pT
        type(vector4_t), intent(in) :: p
        pT = sqrt(p%p(1)**2 + p%p(2)**2)
    end function transverse_part_vector4

```

The scalar longitudinal momentum (assuming the  $z$  axis is longitudinal). Identical to `momentum_z_component`.

```

<Lorentz: public functions>+≡
    public :: longitudinal_part

<Lorentz: interfaces>+≡
    interface longitudinal_part
        module procedure longitudinal_part_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function longitudinal_part_vector4 (p) result (pL)
        real(default) :: pL
        type(vector4_t), intent(in) :: p
        pL = p%p(3)
    end function longitudinal_part_vector4

```

Absolute value of three-momentum

```

<Lorentz: public functions>+≡
    public :: space_part_norm

<Lorentz: interfaces>+≡
    interface space_part_norm
        module procedure space_part_norm_vector4
    end interface

```

```

<Lorentz: procedures>+≡
  elemental function space_part_norm_vector4 (p) result (p3)
    real(default) :: p3
    type(vector4_t), intent(in) :: p
    p3 = sqrt (p%p(1)**2 + p%p(2)**2 + p%p(3)**2)
  end function space_part_norm_vector4

```

The energy (the zeroth component)

```

<Lorentz: public functions>+≡
  public :: energy

<Lorentz: interfaces>+≡
  interface energy
    module procedure energy_vector4
    module procedure energy_vector3
    module procedure energy_real
  end interface

<Lorentz: procedures>+≡
  elemental function energy_vector4 (p) result (E)
    real(default) :: E
    type(vector4_t), intent(in) :: p
    E = p%p(0)
  end function energy_vector4

```

Alternative: The energy corresponding to a given momentum and mass. If the mass is omitted, it is zero

```

<Lorentz: procedures>+≡
  elemental function energy_vector3 (p, mass) result (E)
    real(default) :: E
    type(vector3_t), intent(in) :: p
    real(default), intent(in), optional :: mass
    if (present (mass)) then
      E = sqrt (p**2 + mass**2)
    else
      E = p**1
    end if
  end function energy_vector3

  elemental function energy_real (p, mass) result (E)
    real(default) :: E
    real(default), intent(in) :: p
    real(default), intent(in), optional :: mass
    if (present (mass)) then
      E = sqrt (p**2 + mass**2)
    else
      E = abs (p)
    end if
  end function energy_real

```

The invariant mass of four-momenta. Zero for lightlike, negative for spacelike momenta.

```

<Lorentz: public functions>+≡
  public :: invariant_mass

```

```

<Lorentz: interfaces>+≡
  interface invariant_mass
    module procedure invariant_mass_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function invariant_mass_vector4 (p) result (m)
    real(default) :: m
    type(vector4_t), intent(in) :: p
    real(default) :: msq
    msq = p*p
    if (msq >= 0) then
      m = sqrt (msq)
    else
      m = - sqrt (abs (msq))
    end if
  end function invariant_mass_vector4

```

The invariant mass squared. Zero for lightlike, negative for spacelike momenta.

```

<Lorentz: public functions>+≡
  public :: invariant_mass_squared

<Lorentz: interfaces>+≡
  interface invariant_mass_squared
    module procedure invariant_mass_squared_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function invariant_mass_squared_vector4 (p) result (msq)
    real(default) :: msq
    type(vector4_t), intent(in) :: p
    msq = p*p
  end function invariant_mass_squared_vector4

```

The transverse mass. If the mass squared is negative, this value also is negative.

```

<Lorentz: public functions>+≡
  public :: transverse_mass

<Lorentz: interfaces>+≡
  interface transverse_mass
    module procedure transverse_mass_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function transverse_mass_vector4 (p) result (m)
    real(default) :: m
    type(vector4_t), intent(in) :: p
    real(default) :: msq
    msq = p%p(0)**2 - p%p(1)**2 - p%p(2)**2
    if (msq >= 0) then
      m = sqrt (msq)
    else
      m = - sqrt (abs (msq))
    end if
  end function transverse_mass_vector4

```

The rapidity (defined if particle is massive or  $p_{\perp} > 0$ )

```

<Lorentz: public functions>+≡
    public :: rapidity

<Lorentz: interfaces>+≡
    interface rapidity
        module procedure rapidity_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function rapidity_vector4 (p) result (y)
        real(default) :: y
        type(vector4_t), intent(in) :: p
        y = .5 * log( (energy (p) + longitudinal_part (p)) &
            &          /(energy (p) - longitudinal_part (p)))
    end function rapidity_vector4

```

The pseudorapidity (defined if  $p_{\perp} > 0$ )

```

<Lorentz: public functions>+≡
    public :: pseudorapidity

<Lorentz: interfaces>+≡
    interface pseudorapidity
        module procedure pseudorapidity_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function pseudorapidity_vector4 (p) result (eta)
        real(default) :: eta
        type(vector4_t), intent(in) :: p
        eta = -log( tan (.5 * polar_angle (p)))
    end function pseudorapidity_vector4

```

The rapidity distance (defined if both  $p_{\perp} > 0$ )

```

<Lorentz: public functions>+≡
    public :: rapidity_distance

<Lorentz: interfaces>+≡
    interface rapidity_distance
        module procedure rapidity_distance_vector4
    end interface

<Lorentz: procedures>+≡
    elemental function rapidity_distance_vector4 (p, q) result (dy)
        type(vector4_t), intent(in) :: p, q
        real(default) :: dy
        dy = rapidity (q) - rapidity (p)
    end function rapidity_distance_vector4

```

The pseudorapidity distance (defined if both  $p_{\perp} > 0$ )

```

<Lorentz: public functions>+≡
    public :: pseudorapidity_distance

<Lorentz: interfaces>+≡
    interface pseudorapidity_distance
        module procedure pseudorapidity_distance_vector4
    end interface

```

```

<Lorentz: procedures>+≡
  elemental function pseudorapidity_distance_vector4 (p, q) result (deta)
    real(default) :: deta
    type(vector4_t), intent(in) :: p, q
    deta = pseudorapidity (q) - pseudorapidity (p)
  end function pseudorapidity_distance_vector4

```

The distance on the  $\eta - \phi$  cylinder:

```

<Lorentz: public functions>+≡
  public :: eta_phi_distance

<Lorentz: interfaces>+≡
  interface eta_phi_distance
    module procedure eta_phi_distance_vector4
  end interface

<Lorentz: procedures>+≡
  elemental function eta_phi_distance_vector4 (p, q) result (dr)
    type(vector4_t), intent(in) :: p, q
    real(default) :: dr
    dr = sqrt ( &
      pseudorapidity_distance (p, q)**2 &
      + azimuthal_distance (p, q)**2)
  end function eta_phi_distance_vector4

```

#### 4.2.6 Lorentz transformations

```

<Lorentz: public>+≡
  public :: lorentz_transformation_t

<Lorentz: types>+≡
  type :: lorentz_transformation_t
    private
    real(default), dimension(0:3, 0:3) :: L
    contains
    <Lorentz: lorentz transformation: TBP>
  end type lorentz_transformation_t

```

Output:

```

<Lorentz: public>+≡
  public :: lorentz_transformation_write

<Lorentz: lorentz transformation: TBP>≡
  procedure :: write => lorentz_transformation_write

<Lorentz: procedures>+≡
  subroutine lorentz_transformation_write (L, unit)
    class(lorentz_transformation_t), intent(in) :: L
    integer, intent(in), optional :: unit
    integer :: u
    integer :: i
    u = output_unit (unit); if (u < 0) return
    write (u, 1) "L00 = ", L%L(0,0)
    write (u, 1) "L0j = ", L%L(0,1:3)

```

```

do i = 1, 3
  write (u, 2) "L", i, "0 = ", L%L(i,0)
  write (u, 2) "L", i, "j = ", L%L(i,1:3)
end do
1  format (1x,A,3(1x,ES19.12))
2  format (1x,A,I0,A,3(1x,ES19.12))
end subroutine lorentz_transformation_write

```

Extract all components:

```

<Lorentz: public>+≡
  public :: lorentz_transformation_get_components

<Lorentz: procedures>+≡
  pure function lorentz_transformation_get_components (L) result (a)
    type(lorentz_transformation_t), intent(in) :: L
    real(default), dimension(0:3,0:3) :: a
    a = L%L
  end function lorentz_transformation_get_components

```

#### 4.2.7 Functions of Lorentz transformations

For the inverse, we make use of the fact that  $\Lambda^{\mu\nu}\Lambda_{\mu\rho} = \delta_{\rho}^{\nu}$ . So, lowering the indices and transposing is sufficient.

```

<Lorentz: public functions>+≡
  public :: inverse

<Lorentz: interfaces>+≡
  interface inverse
    module procedure lorentz_transformation_inverse
  end interface

<Lorentz: procedures>+≡
  elemental function lorentz_transformation_inverse (L) result (IL)
    type(lorentz_transformation_t) :: L
    type(lorentz_transformation_t), intent(in) :: L
    IL%L(0,0) = L%L(0,0)
    IL%L(0,1:) = -L%L(1:,0)
    IL%L(1:,0) = -L%L(0,1:)
    IL%L(1:,1:) = transpose(L%L(1:,1:))
  end function lorentz_transformation_inverse

```

#### 4.2.8 Invariants

These are used below. The first array index is varying fastest in FORTRAN; therefore the extra minus in the odd-rank tensor epsilon.

```

<Lorentz: parameters>+≡
  integer, dimension(3,3), parameter :: delta_three = &
    & reshape( source = (/ 1,0,0, 0,1,0, 0,0,1 /), &
    & shape = (/3,3/) )
  integer, dimension(3,3,3), parameter :: epsilon_three = &
    & reshape( source = (/ 0, 0,0, 0,0,-1, 0,1,0,&

```

```

&                0, 0,1,  0,0, 0,  -1,0,0,&
&                0,-1,0,  1,0, 0,   0,0,0 /),&
&                shape = (/3,3,3/) )

```

This could be of some use:

```

<Lorentz: public>+≡
  public :: identity

<Lorentz: parameters>+≡
  type(lorentz_transformation_t), parameter :: &
    & identity = &
    & lorentz_transformation_t ( &
    & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
    &                0._default, 1._default, 0._default, 0._default, &
    &                0._default, 0._default, 1._default, 0._default, &
    &                0._default, 0._default, 0._default, 1._default /),&
    &                shape = (/ 4,4 /) ) )

<Lorentz: public>+≡
  public :: space_reflection

<Lorentz: parameters>+≡
  type(lorentz_transformation_t), parameter :: &
    & space_reflection = &
    & lorentz_transformation_t ( &
    & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
    &                0._default,-1._default, 0._default, 0._default, &
    &                0._default, 0._default,-1._default, 0._default, &
    &                0._default, 0._default, 0._default,-1._default /),&
    &                shape = (/ 4,4 /) ) )

```

#### 4.2.9 Boosts

We build Lorentz transformations from boosts and rotations. In both cases we can supply a three-vector which defines the axis and (hyperbolic) angle. For a boost, this is the vector  $\vec{\beta} = \vec{p}/E$ , such that a particle at rest with mass  $m$  is boosted to a particle with three-vector  $\vec{p}$ . Here, we have

$$\beta = \tanh \chi = p/E, \quad \gamma = \cosh \chi = E/m, \quad \beta\gamma = \sinh \chi = p/m \quad (4.1)$$

```

<Lorentz: public functions>+≡
  public :: boost

<Lorentz: interfaces>+≡
  interface boost
    module procedure boost_from_rest_frame
    module procedure boost_from_rest_frame_vector3
    module procedure boost_generic
    module procedure boost_canonical
  end interface

```

In the first form, the argument is some four-momentum, the space part of which determines a direction, and the associated mass (which is not checked against the four-momentum). The boost vector  $\gamma\vec{\beta}$  is then given by  $\vec{p}/m$ . This boosts from the rest frame of a particle to the current frame. To be explicit, if  $\vec{p}$  is the momentum of a particle and  $m$  its mass,  $L(\vec{p}/m)$  is the transformation that

turns  $(m; \vec{0})$  into  $(E; \vec{p})$ . Conversely, the inverse transformation boosts a vector *into* the rest frame of a particle, in particular  $(E; \vec{p})$  into  $(m; \vec{0})$ .

*(Lorentz: procedures)* +=

```

elemental function boost_from_rest_frame (p, m) result (L)
  type(lorentz_transformation_t) :: L
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: m
  L = boost_from_rest_frame_vector3 (space_part (p), m)
end function boost_from_rest_frame
elemental function boost_from_rest_frame_vector3 (p, m) result (L)
  type(lorentz_transformation_t) :: L
  type(vector3_t), intent(in) :: p
  real(default), intent(in) :: m
  type(vector3_t) :: beta_gamma
  real(default) :: bg2, g, c
  integer :: i,j
  if (m /= 0) then
    beta_gamma = p / m
    bg2 = beta_gamma**2
  else
    bg2 = 0
  end if
  if (bg2 /= 0) then
    g = sqrt(1 + bg2); c = (g-1)/bg2
    L%L(0,0) = g
    L%L(0,1:) = beta_gamma%p
    L%L(1:,0) = L%L(0,1:)
    do i=1,3
      do j=1,3
        L%L(i,j) = delta_three(i,j) + c*beta_gamma%p(i)*beta_gamma%p(j)
      end do
    end do
  else
    L = identity
  end if
end function boost_from_rest_frame_vector3

```

A canonical boost is a boost along one of the coordinate axes, which we may supply as an integer argument. Here,  $\gamma\beta$  is scalar.

*(Lorentz: procedures)* +=

```

elemental function boost_canonical (beta_gamma, k) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: beta_gamma
  integer, intent(in) :: k
  real(default) :: g
  g = sqrt(1 + beta_gamma**2)
  L = identity
  L%L(0,0) = g
  L%L(0,k) = beta_gamma
  L%L(k,0) = L%L(0,k)
  L%L(k,k) = L%L(0,0)
end function boost_canonical

```

Instead of a canonical axis, we can supply an arbitrary axis which need not be normalized. If it is zero, return the unit matrix.



```

(Lorentz: procedures)+≡
elemental function boost_generic (beta_gamma, axis) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: beta_gamma
  type(vector3_t), intent(in) :: axis
  if (any(axis%p/=0)) then
    L = boost_from_rest_frame_vector3 (beta_gamma * axis, axis**1)
  else
    L = identity
  end if
end function boost_generic

```

#### 4.2.10 Rotations

For a rotation, the vector defines the rotation axis, and its length the rotation angle.

```

(Lorentz: public functions)+≡
  public :: rotation

(Lorentz: interfaces)+≡
  interface rotation
    module procedure rotation_generic
    module procedure rotation_canonical
    module procedure rotation_generic_cs
    module procedure rotation_canonical_cs
  end interface

```

If  $\cos \phi$  and  $\sin \phi$  is already known, we do not have to calculate them. Of course, the user has to ensure that  $\cos^2 \phi + \sin^2 \phi = 1$ , and that the given axis **n** is normalized to one. In the second form, the length of **axis** is the rotation angle.

```

(Lorentz: procedures)+≡
elemental function rotation_generic_cs (cp, sp, axis) result (R)
  type(lorentz_transformation_t) :: R
  real(default), intent(in) :: cp, sp
  type(vector3_t), intent(in) :: axis
  integer :: i,j
  R = identity
  do i=1,3
    do j=1,3
      R%L(i,j) = cp*delta_three(i,j) + (1-cp)*axis%p(i)*axis%p(j) &
        & - sp*dot_product(epsilon_three(i,j,:), axis%p)
    end do
  end do
end function rotation_generic_cs
elemental function rotation_generic (axis) result (R)
  type(lorentz_transformation_t) :: R
  type(vector3_t), intent(in) :: axis
  real(default) :: phi
  if (any(axis%p/=0)) then
    phi = abs(axis**1)
    R = rotation_generic_cs (cos(phi), sin(phi), axis/phi)
  else

```

```

        R = identity
    end if
end function rotation_generic

```

Alternatively, give just the angle and label the coordinate axis by an integer.

```

(Lorentz: procedures)+≡
elemental function rotation_canonical_cs (cp, sp, k) result (R)
    type(lorentz_transformation_t) :: R
    real(default), intent(in) :: cp, sp
    integer, intent(in) :: k
    integer :: i,j
    R = identity
    do i=1,3
        do j=1,3
            R%L(i,j) = -sp*epsilon_three(i,j,k)
        end do
        R%L(i,i) = cp
    end do
    R%L(k,k) = 1
end function rotation_canonical_cs
elemental function rotation_canonical (phi, k) result (R)
    type(lorentz_transformation_t) :: R
    real(default), intent(in) :: phi
    integer, intent(in) :: k
    R = rotation_canonical_cs(cos(phi), sin(phi), k)
end function rotation_canonical

```

This is viewed as a method for the first argument (three-vector): Reconstruct the rotation that rotates it into the second three-vector.

```

(Lorentz: public functions)+≡
    public :: rotation_to_2nd

(Lorentz: interfaces)+≡
    interface rotation_to_2nd
        module procedure rotation_to_2nd_generic
        module procedure rotation_to_2nd_canonical
    end interface

(Lorentz: procedures)+≡
elemental function rotation_to_2nd_generic (p, q) result (R)
    type(lorentz_transformation_t) :: R
    type(vector3_t), intent(in) :: p, q
    type(vector3_t) :: a, b, ab
    real(default) :: ct, st
    if (any (p%p /= 0) .and. any (q%p /= 0)) then
        a = direction (p)
        b = direction (q)
        ab = cross_product(a,b)
        ct = a*b;  st = ab**1
        if (st /= 0) then
            R = rotation_generic_cs (ct, st, ab/st)
        else if (ct < 0) then
            R = space_reflection
        else
            R = identity
        end if
    end if

```

```

    else
        R = identity
    end if
end function rotation_to_2nd_generic

```

The same for a canonical axis: The function returns the transformation that rotates the  $k$ -axis into the direction of  $p$ .

```

(Lorentz: procedures)+≡
elemental function rotation_to_2nd_canonical (k, p) result (R)
    type(lorentz_transformation_t) :: R
    integer, intent(in) :: k
    type(vector3_t), intent(in) :: p
    type(vector3_t) :: b, ab
    real(default) :: ct, st
    integer :: i, j
    if (any (p%p /= 0)) then
        b = direction (p)
        ab%p = 0
        do i = 1, 3
            do j = 1, 3
                ab%p(j) = ab%p(j) + b%p(i) * epsilon_three(i,j,k)
            end do
        end do
        ct = b%p(k); st = ab**1
        if (st /= 0) then
            R = rotation_generic_cs (ct, st, ab/st)
        else if (ct < 0) then
            R = space_reflection
        else
            R = identity
        end if
    else
        R = identity
    end if
end function rotation_to_2nd_canonical

```

#### 4.2.11 Composite Lorentz transformations

This function returns the transformation that, given a pair of vectors  $p_{1,2}$ , (a) boosts from the rest frame of the c.m. system (with invariant mass  $m$ ) into the lab frame where  $p_i$  are defined, and (b) turns the given axis (or the canonical vectors  $\pm e_k$ ) in the rest frame into the directions of  $p_{1,2}$  in the lab frame. Note that the energy components are not used; for a consistent result one should have  $(p_1 + p_2)^2 = m^2$ .

```

(Lorentz: public functions)+≡
    public :: transformation

(Lorentz: interfaces)+≡
    interface transformation
        module procedure transformation_rec_generic
        module procedure transformation_rec_canonical
    end interface

```

```

<Lorentz: procedures>+≡
  elemental function transformation_rec_generic (axis, p1, p2, m) result (L)
    type(vector3_t), intent(in) :: axis
    type(vector4_t), intent(in) :: p1, p2
    real(default), intent(in) :: m
    type(lorentz_transformation_t) :: L
    L = boost (p1 + p2, m)
    L = L * rotation_to_2nd (axis, space_part (inverse (L) * p1))
  end function transformation_rec_generic
  elemental function transformation_rec_canonical (k, p1, p2, m) result (L)
    integer, intent(in) :: k
    type(vector4_t), intent(in) :: p1, p2
    real(default), intent(in) :: m
    type(lorentz_transformation_t) :: L
    L = boost (p1 + p2, m)
    L = L * rotation_to_2nd (k, space_part (inverse (L) * p1))
  end function transformation_rec_canonical

```

#### 4.2.12 Applying Lorentz transformations

Multiplying vectors and Lorentz transformations is straightforward.

```

<Lorentz: interfaces>+≡
  interface operator(*)
    module procedure prod_LT_vector4
    module procedure prod_LT_LT
    module procedure prod_vector4_LT
  end interface

<Lorentz: procedures>+≡
  elemental function prod_LT_vector4 (L, p) result (np)
    type(vector4_t) :: np
    type(lorentz_transformation_t), intent(in) :: L
    type(vector4_t), intent(in) :: p
    np%p = matmul (L%L, p%p)
  end function prod_LT_vector4
  elemental function prod_LT_LT (L1, L2) result (NL)
    type(lorentz_transformation_t) :: NL
    type(lorentz_transformation_t), intent(in) :: L1, L2
    NL%L = matmul (L1%L, L2%L)
  end function prod_LT_LT
  elemental function prod_vector4_LT (p, L) result (np)
    type(vector4_t) :: np
    type(vector4_t), intent(in) :: p
    type(lorentz_transformation_t), intent(in) :: L
    np%p = matmul (p%p, L%L)
  end function prod_vector4_LT

```

#### 4.2.13 Special Lorentz transformations

These routines have their application in the generation and extraction of angles in the phase-space sampling routine. Since this part of the program is

time-critical, we calculate the composition of transformations directly instead of multiplying rotations and boosts.

This Lorentz transformation is the composition of a rotation by  $\phi$  around the 3 axis, a rotation by  $\theta$  around the 2 axis, and a boost along the 3 axis:

$$L = B_3(\beta\gamma) R_2(\theta) R_3(\phi) \quad (4.2)$$

Instead of the angles we provide sine and cosine.

```

(Lorentz: public functions)+≡
  public :: LT_compose_r3_r2_b3

(Lorentz: procedures)+≡
  elemental function LT_compose_r3_r2_b3 &
    (cp, sp, ct, st, beta_gamma) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: cp, sp, ct, st, beta_gamma
  real(default) :: gamma
  if (beta_gamma==0) then
    L%L(0,0) = 1
    L%L(1:,0) = 0
    L%L(0,1:) = 0
    L%L(1,1:) = (/ ct*cp, -ct*sp, st /)
    L%L(2,1:) = (/      sp,      cp, 0._default /)
    L%L(3,1:) = (/ -st*cp, st*sp, ct /)
  else
    gamma = sqrt(1 + beta_gamma**2)
    L%L(0,0) = gamma
    L%L(1,0) = 0
    L%L(2,0) = 0
    L%L(3,0) = beta_gamma
    L%L(0,1:) = beta_gamma * (/ -st*cp, st*sp, ct /)
    L%L(1,1:) = (/ ct*cp, -ct*sp, st /)
    L%L(2,1:) = (/      sp,      cp, 0._default /)
    L%L(3,1:) = gamma * (/ -st*cp, st*sp, ct /)
  end if
end function LT_compose_r3_r2_b3

```

Different ordering:

$$L = B_3(\beta\gamma) R_3(\phi) R_2(\theta) \quad (4.3)$$

```

(Lorentz: public functions)+≡
  public :: LT_compose_r2_r3_b3

(Lorentz: procedures)+≡
  elemental function LT_compose_r2_r3_b3 &
    (ct, st, cp, sp, beta_gamma) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: ct, st, cp, sp, beta_gamma
  real(default) :: gamma
  if (beta_gamma==0) then
    L%L(0,0) = 1
    L%L(1:,0) = 0
    L%L(0,1:) = 0
    L%L(1,1:) = (/ ct*cp,      -sp,      st*cp /)
    L%L(2,1:) = (/ ct*sp,      cp,      st*sp /)
  end if
end function LT_compose_r2_r3_b3

```

```

        L%L(3,1:) = (/ -st      , 0._default, ct      /)
    else
        gamma = sqrt(1 + beta_gamma**2)
        L%L(0,0) = gamma
        L%L(1,0) = 0
        L%L(2,0) = 0
        L%L(3,0) = beta_gamma
        L%L(0,1:) = beta_gamma * (/ -st      , 0._default, ct      /)
        L%L(1,1:) =              (/ ct*cp,    -sp,      st*cp /)
        L%L(2,1:) =              (/ ct*sp,     cp,      st*sp /)
        L%L(3,1:) = gamma        * (/ -st      , 0._default, ct      /)
    end if
end function LT_compose_r2_r3_b3

```

This function returns the previous Lorentz transformation applied to an arbitrary four-momentum and extracts the space part of the result:

$$\vec{n} = [B_3(\beta\gamma) R_2(\theta) R_3(\phi) p]_{\text{space part}} \quad (4.4)$$

The second variant applies if there is no rotation

```

<Lorentz: public functions>+≡
    public :: axis_from_p_r3_r2_b3, axis_from_p_b3

<Lorentz: procedures>+≡
    elemental function axis_from_p_r3_r2_b3 &
        (p, cp, sp, ct, st, beta_gamma) result (n)
        type(vector3_t) :: n
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: cp, sp, ct, st, beta_gamma
        real(default) :: gamma, px, py
        px = cp * p%p(1) - sp * p%p(2)
        py = sp * p%p(1) + cp * p%p(2)
        n%p(1) = ct * px + st * p%p(3)
        n%p(2) = py
        n%p(3) = -st * px + ct * p%p(3)
        if (beta_gamma/=0) then
            gamma = sqrt(1 + beta_gamma**2)
            n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
        end if
    end function axis_from_p_r3_r2_b3

    elemental function axis_from_p_b3 (p, beta_gamma) result (n)
        type(vector3_t) :: n
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: beta_gamma
        real(default) :: gamma
        n%p = p%p(1:3)
        if (beta_gamma/=0) then
            gamma = sqrt(1 + beta_gamma**2)
            n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
        end if
    end function axis_from_p_b3

```

#### 4.2.14 Special functions

The standard phase space function:

```
<Lorentz: public functions>+≡
    public :: lambda

<Lorentz: procedures>+≡
    elemental function lambda (m1sq, m2sq, m3sq)
        real(default) :: lambda
        real(default), intent(in) :: m1sq, m2sq, m3sq
        lambda = (m1sq - m2sq - m3sq)**2 - 4*m2sq*m3sq
    end function lambda
```

Return a pair of head-to-head colliding momenta, given the collider energy, particle masses, and optionally the momentum of the c.m. system.

```
<Lorentz: public functions>+≡
    public :: colliding_momenta

<Lorentz: procedures>+≡
    function colliding_momenta (sqrts, m, p_cm) result (p)
        type(vector4_t), dimension(2) :: p
        real(default), intent(in) :: squrts
        real(default), dimension(2), intent(in), optional :: m
        real(default), intent(in), optional :: p_cm
        real(default), dimension(2) :: dmsq
        real(default) :: ch, sh
        real(default), dimension(2) :: E0, p0
        integer, dimension(2), parameter :: sgn = (/1, -1/)
        if (sqrts == 0) then
            call msg_fatal (" Colliding beams: squrts is zero (please set squrts)")
            p = vector4_null; return
        else if (sqrts <= 0) then
            call msg_fatal (" Colliding beams: squrts is negative")
            p = vector4_null; return
        end if
        if (present (m)) then
            dmsq = sgn * (m(1)**2-m(2)**2)
            E0 = (sqrts + dmsq/sqrts) / 2
            if (any (E0 < m)) then
                call msg_fatal &
                    (" Colliding beams: beam energy is less than particle mass")
                p = vector4_null; return
            end if
            p0 = sgn * sqrt (E0**2 - m**2)
        else
            E0 = squrts / 2
            p0 = sgn * E0
        end if
        if (present (p_cm)) then
            sh = p_cm / squrts
            ch = sqrt (1 + sh**2)
            p = vector4_moving (E0 * ch + p0 * sh, E0 * sh + p0 * ch, 3)
        else
            p = vector4_moving (E0, p0, 3)
        end if
```

```
end function colliding_momenta
```

This subroutine is for the purpose of numerical checks and comparisons. The idea is to set a number to zero if it is numerically equivalent with zero. The equivalence is established by comparing with a **tolerance** argument. We implement this for vectors and transformations.

```
<Lorentz: public functions>+≡
  public :: pacify

<Lorentz: interfaces>+≡
  interface pacify
    module procedure pacify_real_default
    module procedure pacify_vector3
    module procedure pacify_vector4
    module procedure pacify_LT
  end interface pacify

<Lorentz: procedures>+≡
  elemental subroutine pacify_real_default (x, tolerance)
    real(default), intent(inout) :: x
    real(default), intent(in) :: tolerance
    if (abs (x) < tolerance) x = 0
  end subroutine pacify_real_default

  elemental subroutine pacify_vector3 (p, tolerance)
    type(vector3_t), intent(inout) :: p
    real(default), intent(in) :: tolerance
    where (abs (p%p) < tolerance) p%p = 0
  end subroutine pacify_vector3

  elemental subroutine pacify_vector4 (p, tolerance)
    type(vector4_t), intent(inout) :: p
    real(default), intent(in) :: tolerance
    where (abs (p%p) < tolerance) p%p = 0
  end subroutine pacify_vector4

  elemental subroutine pacify_LT (LT, tolerance)
    type(lorentz_transformation_t), intent(inout) :: LT
    real(default), intent(in) :: tolerance
    where (abs (LT%L) < tolerance) LT%L = 0
  end subroutine pacify_LT
```



## 4.3 Special Physics functions

Here, we declare functions that are specific for the Standard Model, including QCD: fixed and running  $\alpha_s$ , Catani-Seymour dipole terms, loop functions, etc.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

```

<sm_physics.f90>≡
  <File header>

  module sm_physics

    <Use kinds>
    use constants !NODEP!
    <Use file utils>
    use limits, only: MZ_REF, ALPHA_QCD_MZ_REF, LAMBDA_QCD_REF !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!

    <Standard module head>

    <SM physics: public>

    <SM physics: parameters>

    contains

    <SM physics: procedures>

  end module sm_physics

```

### 4.3.1 Running $\alpha_s$

First we set a reference value for  $\alpha_s(M_Z) = 0.1178$ .

```

<Limits: public parameters>+≡
  real(default), public, parameter :: MZ_REF = 91.188_default
  real(default), public, parameter :: ALPHA_QCD_MZ_REF = 0.1178_default
  real(default), public, parameter :: LAMBDA_QCD_REF = 200.e-3_default

```

Then we define the coefficients of the beta function of QCD (as a reference cf. the Particle Data Group), where  $n_f$  is the number of active flavors in two different schemes:

$$\beta_0 = 11 - \frac{2}{3}n_f \quad (4.5)$$

$$\beta_1 = 51 - \frac{19}{3}n_f \quad (4.6)$$

$$\beta_2 = 2857 - \frac{5033}{9}n_f + \frac{325}{27}n_f^2 \quad (4.7)$$

$$b_0 = \frac{1}{12\pi} (11C_A - 2n_f) \quad (4.8)$$

$$b_1 = \frac{1}{24\pi^2} (17C_A^2 - 5C_An_f - 3C_Fn_f) \quad (4.9)$$

$$b_2 = \frac{1}{(4\pi)^3} \left( \frac{2857}{54} C_A^3 - \frac{1415}{54} C_A^2 n_f - \frac{205}{18} C_A C_F n_f + C_F^2 n_f + \frac{79}{54} C_A n_f^2 + \frac{11}{9} C_F n_f^2 \right) \quad (4.10)$$

$\langle SM \text{ physics: public} \rangle \equiv$

```
public :: beta0, beta1, beta2, coeff_b0, coeff_b1, coeff_b2
```

$\langle SM \text{ physics: procedures} \rangle \equiv$

```
pure function beta0 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta0
  beta0 = 11.0_default - two/three * nf
end function beta0

pure function beta1 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta1
  beta1 = 51.0_default - 19.0_default/three * nf
end function beta1

pure function beta2 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta2
  beta2 = 2857.0_default - 5033.0_default / 9.0_default * &
    nf + 325.0_default/27.0_default * nf**2
end function beta2

pure function coeff_b0 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b0
  coeff_b0 = (11.0_default * CA - two * nf) / (12.0_default * pi)
end function coeff_b0

pure function coeff_b1 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b1
  coeff_b1 = (17.0_default * CA**2 - five * CA * nf - three * CF * nf) / &
    (24.0_default * pi**2)
end function coeff_b1

pure function coeff_b2 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b2
  coeff_b2 = (2857.0_default/54.0_default * CA**3 - &
    1415.0_default/54.0_default * &
    CA**2 * nf - 205.0_default/18.0_default * CA*CF*nf &
    + 79.0_default/54.0_default * CA*nf**2 + &
    11.0_default/9.0_default * CF * nf**2) / (four*pi)**3
end function coeff_b2
```

There should be two versions of running  $\alpha_s$ , one which takes the scale and  $\Lambda_{\text{QCD}}$  as input, and one which takes the scale and e.g.  $\alpha_s(m_Z)$  as input. Here, we take the one which takes the QCD scale and scale as inputs from the PDG book.

```

<SM physics: public>+≡
  public :: running_as, running_as_lam

<SM physics: procedures>+≡
  pure function running_as (scale, al_mz, mz, order, nf) result (ascale)
    real(default), intent(in) :: scale
    real(default), intent(in), optional :: al_mz, nf, mz
    integer, intent(in), optional :: order
    integer :: ord
    real(default) :: az, m_z, as_log, n_f, b0, b1, b2, ascale
    real(default) :: as0, as1
    if (present(mz)) then
      m_z = mz
    else
      m_z = MZ_REF
    end if
    if (present(order)) then
      ord = order
    else
      ord = 0
    end if
    if (present(al_mz)) then
      az = al_mz
    else
      az = ALPHA_QCD_MZ_REF
    end if
    if (present(nf)) then
      n_f = nf
    else
      n_f = 5
    end if
    b0 = coeff_b0 (n_f)
    b1 = coeff_b1 (n_f)
    b2 = coeff_b2 (n_f)
    as_log = one + b0 * az * log(scale**2/m_z**2)
    as0 = az / as_log
    as1 = as0 - as0**2 * b1/b0 * log(as_log)
    select case (ord)
      case (0)
        ascale = as0
      case (1)
        ascale = as1
      case (2)
        ascale = as1 + as0**3 * (b1**2/b0**2 * ((log(as_log))**2 - &
          log(as_log) + as_log - one) - b2/b0 * (as_log - one))
    end select
  end function running_as

  pure function running_as_lam (nf, scale, lambda, order) result (ascale)
    real(default), intent(in) :: nf, scale
    real(default), intent(in), optional :: lambda

```

```

integer, intent(in), optional :: order
real(default) :: lambda_qcd
real(default) :: as0, as1, logmul, b0, b1, b2, ascale
integer :: ord
if (present (lambda)) then
    lambda_qcd = lambda
else
    lambda_qcd = LAMBDA_QCD_REF
end if
if (present(order)) then
    ord = order
else
    ord = 0
end if
b0 = beta0(nf)
b1 = beta1(nf)
b2 = beta2(nf)
logmul = log(scale**2/lambda_qcd**2)
as0 = four*pi / b0 / logmul
as1 = as0 * (one - two* b1 / b0**2 * log(logmul) / logmul)
select case (ord)
case (0)
    ascale = as0
case (1)
    ascale = as1
case (2)
    ascale = as1 + as0 * four * b1**2/b0**4/logmul**2 * &
        ( (log(logmul) - 0.5_default)**2 + &
          b2*b0/8.0_default/b1**2 - five/four)
end select
end function running_as_lam

```

### 4.3.2 Catani-Seymour Parameters

These are fundamental constants of the Catani-Seymour dipole formalism. Since the corresponding parameters for the gluon case depend on the number of flavors which is treated as an argument, there we do have functions and not parameters.

$$\gamma_q = \gamma_{\bar{q}} = \frac{3}{2}C_F \quad \gamma_g = \frac{11}{6}C_A - \frac{2}{3}T_R N_f \quad (4.11)$$

$$K_q = K_{\bar{q}} = \left( \frac{7}{2} - \frac{\pi^2}{6} \right) C_F \quad K_g = \left( \frac{67}{18} - \frac{\pi^2}{6} \right) C_A - \frac{10}{9}T_R N_f \quad (4.12)$$

$\langle SM \text{ physics: parameters} \rangle \equiv$

```

real(kind=default), parameter, public :: gamma_q = three/two * CF, &
    k_q = (7.0_default/two - pi**2/6.0_default) * CF

```

$\langle SM \text{ physics: public} \rangle + \equiv$

```

public :: gamma_g, k_g

```

```

<SM physics: procedures>+≡
  elemental function gamma_g (nf) result (gg)
    real(kind=default), intent(in) :: nf
    real(kind=default) :: gg
    gg = 11.0_default/6.0_default * CA - two/three * TR * nf
  end function gamma_g

  elemental function k_g (nf) result (kg)
    real(kind=default), intent(in) :: nf
    real(kind=default) :: kg
    kg = (67.0_default/18.0_default - pi**2/6.0_default) * CA - &
          10.0_default/9.0_default * TR * nf
  end function k_g

```

### 4.3.3 Mathematical Functions

The dilogarithm. This simplified version is bound to double precision, and restricted to argument values less or equal to unity, so we do not need complex algebra. The wrapper converts it to default precision (which is, of course, a no-op if double=default).

The routine calculates the dilogarithm through mapping on the area where there is a quickly convergent series (adapted from an F77 routine by Hans Kuijf, 1988): Map  $x$  such that  $x$  is not in the neighbourhood of 1. Note that  $|z| = -\ln(1-x)$  is always smaller than 1.10, but  $\frac{1 \cdot 10^{19}}{19!} \text{Bernoulli}_{19} = 2.7 \times 10^{-15}$ .

```

<SM physics: public>+≡
  public :: Li2

<SM physics: procedures>+≡
  function Li2 (x)
    use kinds, only: double !NODEP!
    real(default), intent(in) :: x
    real(default) :: Li2
    Li2 = real( Li2_double (real(x, kind=double)), kind=default)
  end function Li2

<SM physics: procedures>+≡
  function Li2_double (x) result (Li2)
    use kinds, only: double !NODEP!
    real(kind=double), intent(in) :: x
    real(kind=double) :: Li2
    real(kind=double), parameter :: pi2_6 = pi**2/6
    if (abs(1-x) < 1.E-13_double) then
      Li2 = pi2_6
    else if (abs(1-x) < 0.5_double) then
      Li2 = pi2_6 - log(1-x) * log(x) - Li2_restricted (1-x)
    else if (abs(x).gt.1.d0) then
      call msg_bug (" Dilogarithm called outside of defined range.")
    !   Li2 = -pi2_6 - 0.5_default * log(-x) * log(-x) - Li2_restricted (1/x)
    else
      Li2 = Li2_restricted (x)
    end if
  end function Li2_double

```

```

contains
  function Li2_restricted (x) result (Li2)
    real(kind=double), intent(in) :: x
    real(kind=double) :: Li2
    real(kind=double) :: tmp, z, z2
    z = - log (1-x)
    z2 = z**2
    ! Horner's rule for the powers z^3 through z^19
    tmp = 43867._double/798._double
    tmp = tmp * z2 /342._double - 3617._double/510._double
    tmp = tmp * z2 /272._double + 7._double/6._double
    tmp = tmp * z2 /210._double - 691._double/2730._double
    tmp = tmp * z2 /156._double + 5._double/66._double
    tmp = tmp * z2 /110._double - 1._double/30._double
    tmp = tmp * z2 / 72._double + 1._double/42._double
    tmp = tmp * z2 / 42._double - 1._double/30._double
    tmp = tmp * z2 / 20._double + 1._double/6._double
    ! The first three terms of the power series
    Li2 = z2 * z * tmp / 6._double - 0.25_double * z2 + z
  end function Li2_restricted
end function Li2_double

```

#### 4.3.4 Loop Integrals

These functions appear in the calculation of the effective one-loop coupling of a (pseudo)scalar to a vector boson pair.

```

<SM physics: public>+≡
  public :: faux

<SM physics: procedures>+≡
  elemental function faux (x) result (y)
    real(default), intent(in) :: x
    complex(default) :: y
    if (1 <= x) then
      y = asin(sqrt(1/x))**2
    else
      y = - 1/4.0_default * (log((1 + sqrt(1 - x))/ &
        (1 - sqrt(1 - x))) - cmplx (0.0_default, pi, kind=default))**2
    end if
  end function faux

<SM physics: public>+≡
  public :: fonehalf

<SM physics: procedures>+≡
  elemental function fonehalf (x) result (y)
    real(default), intent(in) :: x
    complex(default) :: y
    if (x==0) then
      y = 0
    else
      y = - 2.0_default * x * (1 + (1 - x) * faux(x))
    end if

```

```

end function fonehalf

<SM physics: public>+≡
public :: fonehalf_pseudo

<SM physics: procedures>+≡
function fonehalf_pseudo (x) result (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (x==0) then
    y = 0
  else
    y = - 2.0_default * x * faux(x)
  end if
end function fonehalf_pseudo

<SM physics: public>+≡
public :: fone

<SM physics: procedures>+≡
elemental function fone (x) result (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (x==0) then
    y = 2.0_default
  else
    y = 2.0_default + 3.0_default * x + &
      3.0_default * x * (2.0_default - x) * &
      faux(x)
  end if
end function fone

<SM physics: public>+≡
public :: gaux

<SM physics: procedures>+≡
elemental function gaux (x) result (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (1 <= x) then
    y = sqrt(x - 1) * asin(sqrt(1/x))
  else
    y = sqrt(1 - x) * (log((1 + sqrt(1 - x)) / &
      (1 - sqrt(1 - x)))) - &
      cmplx (0.0_default, pi, kind=default)) / 2.0_default
  end if
end function gaux

<SM physics: public>+≡
public :: tri_i1

```

```

⟨SM physics: procedures⟩+≡
  elemental function tri_i1 (a,b) result (y)
    real(default), intent(in) :: a,b
    complex(default) :: y
    if (a < epsilon(a) .or. b < epsilon(b)) then
      y = 0
    else
      y = a*b/2.0_default/(a-b) + a**2 * b**2/2.0_default/(a-b)**2 * &
        (faux(a) - faux(b)) + &
        a**2 * b/(a-b)**2 * (gaux(a) - gaux(b))
    end if
  end function tri_i1

```

```

⟨SM physics: public⟩+≡
  public :: tri_i2

```

```

⟨SM physics: procedures⟩+≡
  elemental function tri_i2 (a,b) result (y)
    real(default), intent(in) :: a,b
    complex(default) :: y
    if (a < epsilon(a) .or. b < epsilon(b)) then
      y = 0
    else
      y = - a * b / 2.0_default / (a-b) * (faux(a) - faux(b))
    end if
  end function tri_i2

```

#### 4.3.5 More on $\alpha_s$

These functions are for the running of the strong coupling constants,  $\alpha_s$ .

```

⟨SM physics: public⟩+≡
  public :: run_b0

```

```

⟨SM physics: procedures⟩+≡
  elemental function run_b0 (nf) result (bnull)
    integer, intent(in) :: nf
    real(default) :: bnull
    bnull = 33.0_default - 2.0_default * nf
  end function run_b0

```

```

⟨SM physics: public⟩+≡
  public :: run_b1

```

```

⟨SM physics: procedures⟩+≡
  elemental function run_b1 (nf) result (bone)
    integer, intent(in) :: nf
    real(default) :: bone
    bone = 6.0_default * (153.0_default - 19.0_default * nf)/run_b0(nf)**2
  end function run_b1

```

```

⟨SM physics: public⟩+≡
  public :: run_aa

```



```

⟨SM physics: procedures⟩+≡
  elemental function run_aa (nf) result (aaa)
    integer, intent(in) :: nf
    real(default) :: aaa
    aaa = 12.0_default * PI / run_b0(nf)
  end function run_aa

```

```

⟨SM physics: pubic functions⟩≡
  public :: run_bb

```

```

⟨SM physics: procedures⟩+≡
  elemental function run_bb (nf) result (bbb)
    integer, intent(in) :: nf
    real(default) :: bbb
    bbb = run_b1(nf) / run_aa(nf)
  end function run_bb

```

#### 4.3.6 Functions for Catani-Seymour dipoles

For the automated Catani-Seymour dipole subtraction, we need the following functions.

```

⟨SM physics: public⟩+≡
  public :: ff_dipole

```

```

⟨SM physics: procedures⟩+≡
  pure subroutine ff_dipole (v_ijk,y_ijk,p_ij,pp_k,p_i,p_j,p_k)
    type(vector4_t), intent(in) :: p_i, p_j, p_k
    type(vector4_t), intent(out) :: p_ij, pp_k
    real(kind=default), intent(out) :: y_ijk
    real(kind=default) :: z_i
    real(kind=default), intent(out) :: v_ijk
    z_i = (p_i*p_k) / ((p_k*p_j) + (p_k*p_i))
    y_ijk = (p_i*p_j) / ((p_i*p_j) + (p_i*p_k) + (p_j*p_k))
    p_ij = p_i + p_j - y_ijk/(1.0_default - y_ijk) * p_k
    pp_k = (1.0/(1.0_default - y_ijk)) * p_k
    !!! We don't multiply by alpha_s right here:
    v_ijk = 8.0_default * PI * CF * &
      (2.0 / (1.0 - z_i*(1.0 - y_ijk)) - (1.0 + z_i))
  end subroutine ff_dipole

```

```

⟨SM physics: public⟩+≡
  public :: fi_dipole

```

```

⟨SM physics: procedures⟩+≡
  pure subroutine fi_dipole (v_ija,x_ija,p_ij,pp_a,p_i,p_j,p_a)
    type(vector4_t), intent(in) :: p_i, p_j, p_a
    type(vector4_t), intent(out) :: p_ij, pp_a
    real(kind=default), intent(out) :: x_ija
    real(kind=default) :: z_i
    real(kind=default), intent(out) :: v_ija
    z_i = (p_i*p_a) / ((p_a*p_j) + (p_a*p_i))
    x_ija = ((p_i*p_a) + (p_j*p_a) - (p_i*p_j)) &

```

```

      / ((p_i*p_a) + (p_j*p_a))
p_ij = p_i + p_j - (1.0_default - x_ija) * p_a
pp_a = x_ija * p_a
!!! We don't not multiply by alpha_s right here:
v_ija = 8.0_default * PI * CF * &
      (2.0 / (1.0 - z_i + (1.0 - x_ija)) - (1.0 + z_i)) / x_ija
end subroutine fi_dipole

```

```

<SM physics: public>+≡
public :: if_dipole

<SM physics: procedures>+≡
pure subroutine if_dipole (v_kja,u_j,p_aj,pp_k,p_k,p_j,p_a)
  type(vector4_t), intent(in) :: p_k, p_j, p_a
  type(vector4_t), intent(out) :: p_aj, pp_k
  real(kind=default), intent(out) :: u_j
  real(kind=default) :: x_kja
  real(kind=default), intent(out) :: v_kja
  u_j = (p_a*p_j) / ((p_a*p_j) + (p_a*p_k))
  x_kja = ((p_a*p_k) + (p_a*p_j) - (p_j*p_k)) &
    / ((p_a*p_j) + (p_a*p_k))
  p_aj = x_kja * p_a
  pp_k = p_k + p_j - (1.0_default - x_kja) * p_a
  v_kja = 8.0_default * PI * CF * &
    (2.0 / (1.0 - x_kja + u_j) - (1.0 + x_kja)) / x_kja
end subroutine if_dipole

```

This function depends on a variable number of final state particles whose kinematics all get changed by the initial-initial dipole insertion.

```

<SM physics: public>+≡
public :: ii_dipole

<SM physics: procedures>+≡
pure subroutine ii_dipole (v_jab,v_j,p_in,p_out,flag_1or2)
  type(vector4_t), dimension(:), intent(in) :: p_in
  type(vector4_t), dimension(size(p_in)-1), intent(out) :: p_out
  logical, intent(in) :: flag_1or2
  real(kind=default), intent(out) :: v_j
  real(kind=default), intent(out) :: v_jab
  type(vector4_t) :: p_a, p_b, p_j
  type(vector4_t) :: k, kk
  type(vector4_t) :: p_aj
  real(kind=default) :: x_jab
  integer :: i
  !!! flag_1or2 decides whether this a 12 or 21 dipole
  if (flag_1or2) then
    p_a = p_in(1)
    p_b = p_in(2)
  else
    p_b = p_in(1)
    p_a = p_in(2)
  end if
  !!! We assume that the unresolved particle has always the last
  !!! momentum

```

```

p_j = p_in(size(p_in))
x_jab = ((p_a*p_b) - (p_a*p_j) - (p_b*p_j)) / (p_a*p_b)
v_j = (p_a*p_j) / (p_a * p_b)
p_aj = x_jab * p_a
k = p_a + p_b - p_j
kk = p_aj + p_b
do i = 3, size(p_in)-1
  p_out(i) = p_in(i) - 2.0*((k+kk)*p_in(i))/((k+kk)*(k+kk)) * (k+kk) + &
    (2.0 * (k*p_in(i)) / (k*k)) * kk
end do
if (flag_1or2) then
  p_out(1) = p_aj
  p_out(2) = p_b
else
  p_out(1) = p_b
  p_out(2) = p_aj
end if
v_jab = 8.0_default * PI * CF * &
  (2.0 / (1.0 - x_jab) - (1.0 + x_jab)) / x_jab
end subroutine ii_dipole

```

### 4.3.7 Distributions for integrated dipoles and such

The Dirac delta distribution, modified for Monte-Carlo sampling:

```

<SM physics: public>+≡
  public :: delta

<SM physics: procedures>+≡
  elemental function delta (x,eps) result (z)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: z
    if (x > 1.0_default - eps) then
      z = 1.0_default/eps
    else
      z = 0
    end if
  end function delta

```

The  $+$ -distribution,  $P_+(x) = \left(\frac{1}{1-x}\right)_+$ , for the regularization of soft-collinear singularities. The constant part for the Monte-Carlo sampling is the integral over the splitting function divided by the weight for the WHIZARD numerical integration over the interval.

```

<SM physics: public>+≡
  public :: plus_distr

<SM physics: procedures>+≡
  elemental function plus_distr (x,eps) result (plusd)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: plusd
    if (x > (1.0_default - eps)) then
      plusd = log(eps)/eps
    else
      plusd = one/(one-x)
    end if
  end function plus_distr

```

```

end if
end function plus_distr

```

The splitting function in  $D = 4$  dimensions, regularized as  $+$ -distributions if necessary:

$$P^{qq}(x) = P^{\bar{q}q}(x) = C_F \cdot \left( \frac{1+x^2}{1-x} \right)_+ \quad (4.13)$$

$$P^{qg}(x) = P^{\bar{q}g}(x) = C_F \cdot \frac{1+(1-x)^2}{x} \quad (4.14)$$

$$P^{gq}(x) = P^{g\bar{q}}(x) = T_R \cdot [x^2 + (1-x)^2] \quad (4.15)$$

$$P^{gg}(x) = 2C_A \left[ \left( \frac{1}{1-x} \right)_+ + \frac{1-x}{x} - 1 + x(1-x) \right] + \delta(1-x) \left( \frac{11}{6}C_A - \frac{2}{3}N_f T_R \right) \quad (4.16)$$

Since the number of flavors summed over in the gluon splitting function might depend on the physics case under consideration it is implemented as an input variable.

```

⟨SM physics: public⟩+≡
  public :: pqq

⟨SM physics: procedures⟩+≡
  elemental function pqq (x,eps) result (pqqx)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: pqqx
    if (x > (1.0_default - eps)) then
      pqqx = (eps-one)/two + two*log(eps)/eps - three*(eps-one)/eps/two
    else
      pqqx = (one + x**2)/(one-x)
    end if
    pqqx = CF * pqqx
  end function pqq

⟨SM physics: public⟩+≡
  public :: pgq

⟨SM physics: procedures⟩+≡
  elemental function pgq (x) result (pgqx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: pgqx
    pgqx = TR * (x**2 + (one - x)**2)
  end function pgq

⟨SM physics: public⟩+≡
  public :: pqg

⟨SM physics: procedures⟩+≡
  elemental function pqg (x) result (pqgx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: pqgx
    pqgx = CF * (one + (one - x)**2) / x
  end function pqg

```

```

⟨SM physics: public⟩+≡
  public :: pgg
⟨SM physics: procedures⟩+≡
  elemental function pgg (x, nf, eps) result (pggx)
    real(kind=default), intent(in) :: x, nf, eps
    real(kind=default) :: pggx
    pggx = two * CA * ( plus_distr (x, eps) + (one-x)/x - one + &
                        x*(one-x)) + delta (x, eps) * gamma_g(nf)
  end function pgg

```

For the  $qq$  and  $gg$  cases, there exist “regularized” versions of the splitting functions:

$$P_{\text{reg}}^{qq}(x) = -C_F \cdot (1 + x) \quad (4.17)$$

$$P_{\text{reg}}^{gg}(x) = 2C_A \left[ \frac{1-x}{x} - 1 + x(1-x) \right] \quad (4.18)$$

$$(4.19)$$

```

⟨SM physics: public⟩+≡
  public :: pqq_reg
⟨SM physics: procedures⟩+≡
  elemental function pqq_reg (x) result (pqqregx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: pqqregx
    pqqregx = - CF * (one + x)
  end function pqq_reg

```

```

⟨SM physics: public⟩+≡
  public :: pgg_reg
⟨SM physics: procedures⟩+≡
  elemental function pgg_reg (x) result (pggregx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: pgregx
    pgregx = two * CA * ((one - x)/x - one + x*(one - x))
  end function pgg_reg

```

Here, we collect the expressions needed for integrated Catani-Seymour dipoles, and the so-called flavor kernels. We always distinguish between the “ordinary” Catani-Seymour version, and the one including a phase-space slicing parameter,  $\alpha$ .

The standard flavor kernels  $\bar{K}^{ab}$  are:

$$\bar{K}^{qg}(x) = \bar{K}^{\bar{q}g}(x) = P^{qg}(x) \log((1-x)/x) + CF \times x \quad (4.20)$$

$$\bar{K}^{gq}(x) = \bar{K}^{g\bar{q}}(x) = P^{gq}(x) \log((1-x)/x) + TR \times 2x(1-x) \quad (4.21)$$

$$\bar{K}^{qq} = C_F \left[ \left( \frac{2}{1-x} \log \frac{1-x}{x} \right)_+ - (1+x) \log((1-x)/x) + (1-x) \right] - (5 - \pi^2) \cdot C_F \cdot \delta(1-x) \quad (4.22)$$

$$\bar{K}^{gg} = 2C_A \left[ \left( \frac{1}{1-x} \log \frac{1-x}{x} \right)_+ + \left( \frac{1-x}{x} - 1 + x(1-x) \right) \log((1-x)/x) \right] - \delta(1-x) \left[ (f \right. \quad (4.23)$$

```

<SM physics: public>+≡
  public :: kbarqg

<SM physics: procedures>+≡
  function kbarqg (x) result (kbarqgx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: kbarqgx
    kbarqgx = pqg(x) * log((one-x)/x) + CF * x
  end function kbarqg

<SM physics: public>+≡
  public :: kbargq

<SM physics: procedures>+≡
  function kbargq (x) result (kbargqx)
    real(kind=default), intent(in) :: x
    real(kind=default) :: kbargqx
    kbargqx = pqg(x) * log((one-x)/x) + two * TR * x * (one - x)
  end function kbargq

<SM physics: public>+≡
  public :: kbarqq

<SM physics: procedures>+≡
  function kbarqq (x,eps) result (kbarqqx)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: kbarqqx
    kbarqqx = CF*(log_plus_distr(x,eps) - (one+x) * log((one-x)/x) + (one - &
      x) - (five - pi**2) * delta(x,eps))
  end function kbarqq

<SM physics: public>+≡
  public :: kbargg

<SM physics: procedures>+≡
  function kbargg (x,eps,nf) result (kbarggx)
    real(kind=default), intent(in) :: x, eps, nf
    real(kind=default) :: kbarggx
    kbarggx = CA * (log_plus_distr(x,eps) + two * ((one-x)/x - one + &
      x*(one-x) * log((1-x)/x))) - delta(x,eps) * &
      ((50.0_default/9.0_default - pi**2) * CA - &
      16.0_default/9.0_default * TR * nf)
  end function kbargg

```

The  $\tilde{K}$  are used when two identified hadrons participate:

$$\tilde{K}^{ab}(x) = P_{\text{reg}}^{ab}(x) \cdot \log(1-x) + \delta^{ab} \mathbf{T}_a^2 \left[ \left( \frac{2}{1-x} \log(1-x) \right)_+ - \frac{\pi^2}{3} \delta(1-x) \right] \quad (4.24)$$

```

<SM physics: public>+≡
  public :: ktildeqq

```

```

⟨SM physics: procedures⟩+≡
function ktildeqq (x,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqq_reg (x) * log(one-x) + CF * ( - log2_plus_distr (x,eps) &
    - pi**2/three * delta(x,eps))
end function ktildeqq

```

```

⟨SM physics: public⟩+≡
public :: ktildeqq

```

```

⟨SM physics: procedures⟩+≡
function ktildeqq (x,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqg (x) * log(one-x)
end function ktildeqq

```

```

⟨SM physics: public⟩+≡
public :: ktildeqq

```

```

⟨SM physics: procedures⟩+≡
function ktildeqq (x,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeqqx
  ktildeqqx = pgq (x) * log(one-x)
end function ktildeqq

```

```

⟨SM physics: public⟩+≡
public :: ktildeqq

```

```

⟨SM physics: procedures⟩+≡
function ktildegg (x,eps) result (ktildeggx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeggx
  ktildeggx = pgg_reg (x) * log(one-x) + CA * ( - &
    log2_plus_distr (x,eps) - pi**2/three * delta(x,eps))
end function ktildegg

```

The insertion operator might not be necessary for a GOLEM interface but is demanded by the Les Houches NLO accord. It is a three-dimensional array, where the index always gives the inverse power of the DREG expansion parameter,  $\epsilon$ .

```

⟨SM physics: public⟩+≡
public :: insert_q

```

```

⟨SM physics: procedures⟩+≡
pure function insert_q ()
  real(kind=default), dimension(0:2) :: insert_q
  insert_q(0) = gamma_q + k_q - pi**2/three * CF
  insert_q(1) = gamma_q
  insert_q(2) = CF
end function insert_q

```

```

⟨SM physics: public⟩+≡
  public :: insert_g

⟨SM physics: procedures⟩+≡
  pure function insert_g (nf)
    real(kind=default), intent(in) :: nf
    real(kind=default), dimension(0:2) :: insert_g
    insert_g(0) = gamma_g (nf) + k_g (nf) - pi**2/three * CA
    insert_g(1) = gamma_g (nf)
    insert_g(2) = CA
  end function insert_g

```

For better convergence, one can exclude regions of phase space with a slicing parameter from the dipole subtraction procedure. First of all, the  $K$  functions get modified:

$$K_i(\alpha) = K_i - \mathbf{T}_i^2 \log^2 \alpha + \gamma_i(\alpha - 1 - \log \alpha) \quad (4.25)$$

```

⟨SM physics: public⟩+≡
  public :: k_q_al, k_g_al

⟨SM physics: procedures⟩+≡
  pure function k_q_al (alpha)
    real(kind=default), intent(in) :: alpha
    real(kind=default) :: k_q_al
    k_q_al = k_q - CF * (log(alpha))**2 + gamma_q * &
      (alpha - one - log(alpha))
  end function k_q_al

  pure function k_g_al (alpha, nf)
    real(kind=default), intent(in) :: alpha, nf
    real(kind=default) :: k_g_al
    k_g_al = k_g (nf) - CA * (log(alpha))**2 + gamma_g (nf) * &
      (alpha - one - log(alpha))
  end function k_g_al

```

The  $+$ -distribution, but with a phase-space slicing parameter,  $\alpha$ ,  $P_{1-\alpha}(x) = \left(\frac{1}{1-x}\right)_{1-x}$ . Since we need the fatal error message here, this function cannot be elemental.

```

⟨SM physics: public⟩+≡
  public :: plus_distr_al

⟨SM physics: procedures⟩+≡
  function plus_distr_al (x,alpha,eps) result (plusd_al)
    real(kind=default), intent(in) :: x, eps, alpha
    real(kind=default) :: plusd_al
    if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
      call msg_fatal ('sm_physics, plus_distr_al: alpha and epsilon chosen wrongly')
    elseif (x < (1.0_default - alpha)) then
      plusd_al = 0
    else if (x > (1.0_default - eps)) then
      plusd_al = log(eps/alpha)/eps
    else
      plusd_al = one/(one-x)
    end if
  end function

```



```
end function plus_distr_al
```

Introducing phase-space slicing parameters, these flavor kernels become: The standard flavor kernels  $\bar{K}^{ab}$  are:

$$\bar{K}_\alpha^{qg}(x) = \bar{K}_\alpha^{\bar{q}g}(x) = P^{qg}(x) \log(\alpha(1-x)/x) + CF \times x \quad (4.26)$$

$$\bar{K}_\alpha^{gq}(x) = \bar{K}_\alpha^{g\bar{q}}(x) = P^{gq}(x) \log(\alpha(1-x)/x) + TR \times 2x(1-x) \quad (4.27)$$

$$\bar{K}_\alpha^{qq} = C_F(1-x) + P_{\text{reg}}^{qq}(x) \log \frac{\alpha(1-x)}{x} + C_F \delta(1-x) \log^2 \alpha + C_F \left( \frac{2}{1-x} \log \frac{1-x}{x} \right)_+ - \left( \gamma_q - \right) \quad (4.28)$$

```

⟨SM physics: public⟩+≡
public :: kbarqg_al

⟨SM physics: procedures⟩+≡
function kbarqg_al (x,alpha,eps) result (kbarqgx)
  real(kind=default), intent(in) :: x, alpha, eps
  real(kind=default) :: kbarqgx
  kbarqgx = pqg (x) * log(alpha*(one-x)/x) + CF * x
end function kbarqg_al

⟨SM physics: public⟩+≡
public :: kbargq_al

⟨SM physics: procedures⟩+≡
function kbargq_al (x,alpha,eps) result (kbargqx)
  real(kind=default), intent(in) :: x, alpha, eps
  real(kind=default) :: kbargqx
  kbargqx = pgq (x) * log(alpha*(one-x)/x) + two * TR * x * (one-x)
end function kbargq_al

⟨SM physics: public⟩+≡
public :: kbarqq_al

⟨SM physics: procedures⟩+≡
function kbarqq_al (x,alpha,eps) result (kbarqqx)
  real(kind=default), intent(in) :: x, alpha, eps
  real(kind=default) :: kbarqqx
  kbarqqx = CF * (one - x) + pqq_reg(x) * log(alpha*(one-x)/x) &
    + CF * log_plus_distr(x,eps) &
    - (gamma_q + k_q_al(alpha) - CF * &
      five/6.0_default * pi**2 - CF * (log(alpha))**2) * &
      delta(x,eps) + &
      CF * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
  if (x < (one-alpha)) then
    kbarqqx = kbarqqx - CF * two/(one-x) * log((two-x)/(one-x))
  end if
end function kbarqq_al

⟨SM physics: public⟩+≡
public :: kbargg_al

```

```

⟨SM physics: procedures⟩+≡
function kbargg_al (x,alpha,eps,nf) result (kbarggx)
  real(kind=default), intent(in) :: x, alpha, eps, nf
  real(kind=default) :: kbarggx
  kbarggx = pgg_reg(x) * log(alpha*(one-x)/x) &
    + CA * log_plus_distr(x,eps) &
    - (gamma_g(nf) + k_g_al(alpha,nf) - CA * &
      five/6.0_default * pi**2 - CA * (log(alpha))**2) * &
      delta(x,eps) + &
      CA * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
  if (x < (one-alpha)) then
    kbarggx = kbarggx - CA * two/(one-x) * log((two-x)/(one-x))
  end if
end function kbargg_al

```

The  $\tilde{K}$  flavor kernels in the presence of a phase-space slicing parameter, are:

$$\tilde{K}^{ab}(x, \alpha) = P^{qq, \text{reg}}(x) \log \frac{1-x}{\alpha} + \dots \quad (4.29)$$

```

⟨SM physics: public⟩+≡
public :: ktildeqq_al

⟨SM physics: procedures⟩+≡
function ktildeqq_al (x,alpha,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps, alpha
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqq_reg(x) * log((one-x)/alpha) + CF*( &
    - log2_plus_distr_al(x,alpha,eps) - Pi**2/three * delta(x,eps) &
    + (one+x**2)/(one-x) * log(min(one,(alpha/(one-x)))) &
    + two/(one-x) * log((one+alpha-x)/alpha))
  if (x > (one-alpha)) then
    ktildeqqx = ktildeqqx - CF*two/(one-x)*log(two-x)
  end if
end function ktildeqq_al

```

This is a logarithmic  $+$ -distribution,  $\left(\frac{\log((1-x)/x)}{1-x}\right)_+$ . For the sampling, we need the integral over this function over the incomplete sampling interval  $[0, 1 - \epsilon]$ , which is  $\log^2(x) + 2Li_2(x) - \frac{\pi^2}{3}$ . As this function is negative definite for  $\epsilon > 0.1816$ , we take a hard upper limit for that sampling parameter, irrespective of the fact what the user chooses.

```

⟨SM physics: public⟩+≡
public :: log_plus_distr

⟨SM physics: procedures⟩+≡
function log_plus_distr (x,eps) result (lpd)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: lpd, eps2
  eps2 = min (eps, 0.1816_default)
  if (x > (1.0_default - eps2)) then
    lpd = ((log(eps2))**2 + two*Li2(eps2) - pi**2/three)/eps2
  else
    lpd = two*log((one-x)/x)/(one-x)
  end if
end function log_plus_distr

```

```

end function log_plus_distr

Logarithmic +-distribution,  $2 \left( \frac{\log(1/(1-x))}{1-x} \right)_+$ .
⟨SM physics: public⟩+≡
  public :: log2_plus_distr

⟨SM physics: procedures⟩+≡
  function log2_plus_distr (x,eps) result (lpd)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: lpd
    if (x > (1.0_default - eps)) then
      lpd = - (log(eps))**2/eps
    else
      lpd = two*log(one/(one-x))/(one-x)
    end if
  end function log2_plus_distr

Logarithmic +-distribution with phase-space slicing parameter,  $2 \left( \frac{\log(1/(1-x))}{1-x} \right)_{1-\alpha}$ .
⟨SM physics: public⟩+≡
  public :: log2_plus_distr_al

⟨SM physics: procedures⟩+≡
  function log2_plus_distr_al (x,alpha,eps) result (lpd_al)
    real(kind=default), intent(in) :: x, eps, alpha
    real(kind=default) :: lpd_al
    if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
      call msg_fatal ('alpha and epsilon chosen wrongly')
    elseif (x < (one - alpha)) then
      lpd_al = 0
    elseif (x > (1.0_default - eps)) then
      lpd_al = - ((log(eps))**2 - (log(alpha))**2)/eps
    else
      lpd_al = two*log(one/(one-x))/(one-x)
    end if
  end function log2_plus_distr_al

```

## 4.4 QCD Coupling

We provide various distinct implementations of the QCD coupling. In this module, we define an abstract data type and three implementations: fixed, running with  $\alpha_s(M_Z)$  as input, and running with  $\Lambda_{\text{QCD}}$  as input. We use the functions defined above in the module `sm_physics` but provide a common interface. Later modules may define additional implementations.

```

⟨sm_qcd.f90⟩≡
  ⟨File header⟩

  module sm_qcd

    ⟨Use kinds⟩
    ⟨Use file utils⟩

```

```

    use limits, only: MZ_REF, ALPHA_QCD_MZ_REF, LAMBDA_QCD_REF !NODEP!
    use diagnostics !NODEP!
    use unit_tests

    use sm_physics !NODEP!

    <Standard module head>

    <SM qcd: public>

    <SM qcd: types>

    <SM qcd: interfaces>

    contains

    <SM qcd: procedures>

    <SM qcd: tests>

    end module sm_qcd

```

#### 4.4.1 Coupling: Abstract Data Type

This is the abstract version of the QCD coupling implementation.

```

<SM qcd: public>≡
    public :: alpha_qcd_t

<SM qcd: types>≡
    type, abstract :: alpha_qcd_t
    contains
    <SM qcd: alpha qcd: TBP>
    end type alpha_qcd_t

```

There must be an output routine.

```

<SM qcd: alpha qcd: TBP>≡
    procedure (alpha_qcd_write), deferred :: write

<SM qcd: interfaces>≡
    abstract interface
        subroutine alpha_qcd_write (object, unit)
            import
            class(alpha_qcd_t), intent(in) :: object
            integer, intent(in), optional :: unit
        end subroutine alpha_qcd_write
    end interface

```

This method computes the running coupling, given a certain scale. All parameters (reference value, order of the approximation, etc.) must be set before calling this.

```

<SM qcd: alpha qcd: TBP>+≡
    procedure (alpha_qcd_get), deferred :: get

```

```

⟨SM qcd: interfaces⟩+≡
  abstract interface
    function alpha_qcd_get (alpha_qcd, scale) result (alpha)
      import
      class(alpha_qcd_t), intent(in) :: alpha_qcd
      real(default), intent(in) :: scale
      real(default) :: alpha
    end function alpha_qcd_get
  end interface

```

#### 4.4.2 Fixed Coupling

In this version, the  $\alpha_s$  value is fixed, the `scale` argument of the `get` method is ignored. There is only one parameter, the value. By default, this is the value at  $M_Z$ .

```

⟨SM qcd: public⟩+≡
  public :: alpha_qcd_fixed_t

⟨SM qcd: types⟩+≡
  type, extends (alpha_qcd_t) :: alpha_qcd_fixed_t
    real(default) :: val = ALPHA_QCD_MZ_REF
  contains
    ⟨SM qcd: alpha qcd fixed: TBP⟩
  end type alpha_qcd_fixed_t

```

Output.

```

⟨SM qcd: alpha qcd fixed: TBP⟩≡
  procedure :: write => alpha_qcd_fixed_write

⟨SM qcd: procedures⟩≡
  subroutine alpha_qcd_fixed_write (object, unit)
    class(alpha_qcd_fixed_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A)") "QCD parameters (fixed coupling):"
    write (u, "(5x,A,ES12.5)") "alpha = ", object%val
  end subroutine alpha_qcd_fixed_write

```

Calculation: the scale is ignored in this case.

```

⟨SM qcd: alpha qcd fixed: TBP⟩+≡
  procedure :: get => alpha_qcd_fixed_get

⟨SM qcd: procedures⟩+≡
  function alpha_qcd_fixed_get (alpha_qcd, scale) result (alpha)
    class(alpha_qcd_fixed_t), intent(in) :: alpha_qcd
    real(default), intent(in) :: scale
    real(default) :: alpha
    alpha = alpha_qcd%val
  end function alpha_qcd_fixed_get

```

### 4.4.3 Running Coupling

In this version, the  $\alpha_s$  value runs relative to the value at a given reference scale. There are two parameters: the value of this scale (default:  $M_Z$ ), the value of  $\alpha_s$  at this scale, and the number of effective flavors. Furthermore, we have the order of the approximation.

```

<SM qcd: public>+≡
    public :: alpha_qcd_from_scale_t

<SM qcd: types>+≡
    type, extends (alpha_qcd_t) :: alpha_qcd_from_scale_t
        real(default) :: mu_ref = MZ_REF
        real(default) :: ref = ALPHA_QCD_MZ_REF
        integer :: order = 0
        integer :: nf = 5
    contains
        <SM qcd: alpha qcd from scale: TBP>
    end type alpha_qcd_from_scale_t

```

Output.

```

<SM qcd: alpha qcd from scale: TBP>≡
    procedure :: write => alpha_qcd_from_scale_write

<SM qcd: procedures>+≡
    subroutine alpha_qcd_from_scale_write (object, unit)
        class(alpha_qcd_from_scale_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(3x,A)") "QCD parameters (running coupling):"
        write (u, "(5x,A,ES12.5)") "Scale mu = ", object%mu_ref
        write (u, "(5x,A,ES12.5)") "alpha(mu) = ", object%ref
        write (u, "(5x,A,I0)") "LL order = ", object%order
        write (u, "(5x,A,I0)") "N(flv) = ", object%nf
    end subroutine alpha_qcd_from_scale_write

```

Calculation: here, we call the function for running  $\alpha_s$  that was defined in `sm_physics` above. The function does not take into account thresholds, so the number of flavors should be the correct one for the chosen scale. Normally, this should be the  $Z$  boson mass.

```

<SM qcd: alpha qcd from scale: TBP>+≡
    procedure :: get => alpha_qcd_from_scale_get

<SM qcd: procedures>+≡
    function alpha_qcd_from_scale_get (alpha_qcd, scale) result (alpha)
        class(alpha_qcd_from_scale_t), intent(in) :: alpha_qcd
        real(default), intent(in) :: scale
        real(default) :: alpha
        alpha = running_as (scale, &
            alpha_qcd%ref, alpha_qcd%mu_ref, alpha_qcd%order, &
            real (alpha_qcd%nf, kind=default))
    end function alpha_qcd_from_scale_get

```

#### 4.4.4 Running Coupling, determined by $\Lambda_{\text{QCD}}$

In this version, the input are the value  $\Lambda_{\text{QCD}}$  and the order of the approximation.

```

<SM qcd: public>+≡
    public :: alpha_qcd_from_lambda_t
<SM qcd: types>+≡
    type, extends (alpha_qcd_t) :: alpha_qcd_from_lambda_t
        real(default) :: lambda = LAMBDA_QCD_REF
        integer :: order = 0
        integer :: nf = 5
    contains
        <SM qcd: alpha qcd from lambda: TBP>
        end type alpha_qcd_from_lambda_t

```

Output.

```

<SM qcd: alpha qcd from lambda: TBP>≡
    procedure :: write => alpha_qcd_from_lambda_write
<SM qcd: procedures>+≡
    subroutine alpha_qcd_from_lambda_write (object, unit)
        class(alpha_qcd_from_lambda_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(3x,A)") "QCD parameters (Lambda_QCD as input):"
        write (u, "(5x,A,ES12.5)") "Lambda_QCD = ", object%lambda
        write (u, "(5x,A,I0)") "LL order = ", object%order
        write (u, "(5x,A,I0)") "N(flv) = ", object%nf
    end subroutine alpha_qcd_from_lambda_write

```

Calculation: here, we call the second function for running  $\alpha_s$  that was defined in `sm_physics` above. The  $\Lambda$  value should be the one that is appropriate for the chosen number of effective flavors. Again, thresholds are not incorporated.

```

<SM qcd: alpha qcd from lambda: TBP>+≡
    procedure :: get => alpha_qcd_from_lambda_get
<SM qcd: procedures>+≡
    function alpha_qcd_from_lambda_get (alpha_qcd, scale) result (alpha)
        class(alpha_qcd_from_lambda_t), intent(in) :: alpha_qcd
        real(default), intent(in) :: scale
        real(default) :: alpha
        alpha = running_as_lam (real (alpha_qcd%nf, kind=default), scale, &
            alpha_qcd%lambda, alpha_qcd%order)
    end function alpha_qcd_from_lambda_get

```

#### 4.4.5 Wrapper type

We could get along with a polymorphic QCD type, but a monomorphic wrapper type with a polymorphic component is easier to handle and probably safer (w.r.t. compiler bugs). However, we keep the object transparent, so we can set the type-specific parameters directly (by a `dispatch` routine).

```

<SM qcd: public>+≡

```

```

    public :: qcd_t
  <SM qcd: types>+≡
    type :: qcd_t
      class(alpha_qcd_t), allocatable :: alpha
      contains
      <SM qcd: qcd: TBP>
    end type qcd_t

```

Output. We first print the polymorphic `alpha` which contains a headline, then any extra components.

```

  <SM qcd: qcd: TBP>≡
    procedure :: write => qcd_write
  <SM qcd: procedures>+≡
    subroutine qcd_write (qcd, unit)
      class(qcd_t), intent(in) :: qcd
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (allocated (qcd%alpha)) then
        call qcd%alpha%write (u)
      else
        write (u, "(3x,A)") "QCD parameters (coupling undefined)"
      end if
    end subroutine qcd_write

```

#### 4.4.6 Unit tests

```

  <SM qcd: public>+≡
    public :: sm_qcd_test
  <SM qcd: tests>≡
    subroutine sm_qcd_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    <SM qcd: execute tests>
    end subroutine sm_qcd_test

```

#### QCD Coupling

We check two different implementations of the abstract QCD coupling.

```

  <SM qcd: execute tests>≡
    call test (sm_qcd_1, "sm_qcd_1", &
      "running alpha_s", &
      u, results)
  <SM qcd: tests>+≡
    subroutine sm_qcd_1 (u)
      integer, intent(in) :: u
      type(qcd_t) :: qcd

```



```

write (u, "(A)")  "* Test output: sm_qcd_1"
write (u, "(A)")  "* Purpose: compute running alpha_s"
write (u, "(A)")

write (u, "(A)")  "* Fixed:"
write (u, "(A)")

allocate (alpha_qcd_fixed_t :: qcd%alpha)

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)
deallocate (qcd%alpha)

write (u, "(A)")  "* Running from MZ (LO):"
write (u, "(A)")

allocate (alpha_qcd_from_scale_t :: qcd%alpha)

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
    alpha%order = 1
end select

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NNLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
    alpha%order = 2
end select

```

```

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz) =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

deallocate (qcd%alpha)
write (u, "(A)")  "** Running from Lambda_QCD (LO):"
write (u, "(A)")

allocate (alpha_qcd_from_lambda_t :: qcd%alpha)

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz) =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "** Running from Lambda_QCD (NLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_lambda_t)
    alpha%order = 1
end select

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz) =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "** Running from Lambda_QCD (NNLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_lambda_t)
    alpha%order = 2
end select

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)") "alpha_s (mz) =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)") "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)

```

```
write (u, "(A)")  
write (u, "(A)")  "* Test output end: sm_qcd_1"  
  
end subroutine sm_qcd_1
```

## Chapter 5

# Physics Analysis

This part contains the structures and tools that are necessary for defining parameters, particle sets, analysis objects such as histograms, and expressions that deal with them.

These are the modules:

**analysis** Observables, histograms, and plots.

**pdg\_arrays** Useful for particle aliases (e.g., 'quark' for  $u, d, s$  etc.)

**prt\_lists** Particle lists/arrays used for analyzing events.

**variables** Store values of various kind, used by expressions and accessed by the command interface.

**expressions** Expressions of values of all kinds. Includes the API for recording analysis data.

## 5.1 Analysis tools

This module defines structures useful for data analysis. These include observables, histograms, and plots.

Observables are quantities that are calculated and summed up event by event. At the end, one can compute the average and error.

Histograms have their bins in addition to the observable properties. Histograms are usually written out in tables and displayed graphically.

In plots, each record creates its own entry in a table. This can be used for scatter plots if called event by event, or for plotting dependencies on parameters if called once per integration run.

Graphs are container for histograms and plots, which carry their own graphics options.

The type layout is still somewhat obfuscated. This would become much simpler if type extension could be used.

```
<analysis.f90>≡
  <File header>

  module analysis

    <Use kinds>
    <Use strings>
    use limits, only: HISTOGRAM_HEAD_FORMAT, HISTOGRAM_DATA_FORMAT !NODEP!
    use limits, only: HISTOGRAM_INTG_FORMAT !NODEP!
    <Use file utils>
    use diagnostics !NODEP!
    use os_interface
    use ifiles

    <Standard module head>

    <Analysis: public>

    <Analysis: parameters>

    <Analysis: types>

    <Analysis: interfaces>

    <Analysis: variables>

    contains

    <Analysis: procedures>

  end module analysis
```

### 5.1.1 Output formats

These formats share a common field width (alignment).

```
<Limits: public parameters>+≡
  character(*), parameter, public :: HISTOGRAM_HEAD_FORMAT = "1x,A13,1x"
```

```

character(*), parameter, public :: HISTOGRAM_INTG_FORMAT = "3x,I9,3x"
character(*), parameter, public :: HISTOGRAM_DATA_FORMAT = "1PG15.8"

```

### 5.1.2 Graph options

These parameters are used for displaying data. They apply to a whole graph, which may contain more than one plot element.

The GAMELAN code chunks are part of both `graph_options` and `drawing_options`. The `drawing_options` copy is used in histograms and plots, also as graph elements. The `graph_options` copy is used for `graph` objects as a whole. Both copies are usually identical.

```

<Analysis: public>≡
    public :: graph_options_t

<Analysis: types>≡
    type :: graph_options_t
        private
        type(string_t) :: id
        type(string_t) :: title
        type(string_t) :: description
        type(string_t) :: x_label
        type(string_t) :: y_label
        integer :: width_mm = 130
        integer :: height_mm = 90
        logical :: x_log = .false.
        logical :: y_log = .false.
        real(default) :: x_min = 0
        real(default) :: x_max = 1
        real(default) :: y_min = 0
        real(default) :: y_max = 1
        logical :: x_min_set = .false.
        logical :: x_max_set = .false.
        logical :: y_min_set = .false.
        logical :: y_max_set = .false.
        type(string_t) :: gmlcode_bg
        type(string_t) :: gmlcode_fg
    end type graph_options_t

```

Initialize the record, all strings are empty. The limits are undefined.

```

<Analysis: public>+≡
    public :: graph_options_init

<Analysis: procedures>≡
    subroutine graph_options_init (graph_options)
        type(graph_options_t), intent(out) :: graph_options
        graph_options%id = ""
        graph_options%title = ""
        graph_options%description = ""
        graph_options%x_label = ""
        graph_options%y_label = ""
        graph_options%gmlcode_bg = ""
        graph_options%gmlcode_fg = ""
    end subroutine graph_options_init

```

```
end subroutine graph_options_init
```

Set individual options.

*<Analysis: public>+≡*

```
public :: graph_options_set
```

*<Analysis: procedures>+≡*

```
subroutine graph_options_set (graph_options, id, &
    title, description, x_label, y_label, width_mm, height_mm, &
    x_log, y_log, x_min, x_max, y_min, y_max, &
    gmlcode_bg, gmlcode_fg)
    type(graph_options_t), intent(inout) :: graph_options
    type(string_t), intent(in), optional :: id
    type(string_t), intent(in), optional :: title
    type(string_t), intent(in), optional :: description
    type(string_t), intent(in), optional :: x_label, y_label
    integer, intent(in), optional :: width_mm, height_mm
    logical, intent(in), optional :: x_log, y_log
    real(default), intent(in), optional :: x_min, x_max, y_min, y_max
    type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
    if (present (id)) graph_options%id = id
    if (present (title)) graph_options%title = title
    if (present (description)) graph_options%description = description
    if (present (x_label)) graph_options%x_label = x_label
    if (present (y_label)) graph_options%y_label = y_label
    if (present (width_mm)) graph_options%width_mm = width_mm
    if (present (height_mm)) graph_options%height_mm = height_mm
    if (present (x_log)) graph_options%x_log = x_log
    if (present (y_log)) graph_options%y_log = y_log
    if (present (x_min)) graph_options%x_min = x_min
    if (present (x_max)) graph_options%x_max = x_max
    if (present (y_min)) graph_options%y_min = y_min
    if (present (y_max)) graph_options%y_max = y_max
    if (present (x_min)) graph_options%x_min_set = .true.
    if (present (x_max)) graph_options%x_max_set = .true.
    if (present (y_min)) graph_options%y_min_set = .true.
    if (present (y_max)) graph_options%y_max_set = .true.
    if (present (gmlcode_bg)) graph_options%gmlcode_bg = gmlcode_bg
    if (present (gmlcode_fg)) graph_options%gmlcode_fg = gmlcode_fg
end subroutine graph_options_set
```

Write a L<sup>A</sup>T<sub>E</sub>X header/footer for the analysis file.

*<Analysis: procedures>+≡*

```
subroutine graph_options_write_tex_header (gro, unit)
    type(graph_options_t), intent(in) :: gro
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (gro%title /= "") then
        write (u, "(A)")
        write (u, "(A)") "\section{" // char (gro%title) // "}"
    else
        write (u, "(A)") "\section{" // char (quote_underscore (gro%id)) // "}"
    end if
end subroutine graph_options_write_tex_header
```

```

end if
if (gro%description /= "") then
  write (u, "(A)") char (gro%description)
  write (u, *)
  write (u, "(A)") "\vspace*{\baselineskip}"
end if
write (u, "(A)") "\vspace*{\baselineskip}"
write (u, "(A)") "\unitlength 1mm"
write (u, "(A,I0,',',I0,A)") &
  "\begin{gmlgraph*}(", &
  gro%width_mm, gro%height_mm, &
  ")[dat]"
end subroutine graph_options_write_tex_header

subroutine graph_options_write_tex_footer (gro, unit)
  type(graph_options_t), intent(in) :: gro
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(A)") "\end{gmlgraph*}"
end subroutine graph_options_write_tex_footer

```

Return the analysis object ID.

```

<Analysis: procedures>+≡
function graph_options_get_id (gro) result (id)
  type(string_t) :: id
  type(graph_options_t), intent(in) :: gro
  id = gro%id
end function graph_options_get_id

```

Create an appropriate setup command (linear/log).

```

<Analysis: procedures>+≡
function graph_options_get_gml_setup (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  type(string_t) :: x_str, y_str
  if (gro%x_log) then
    x_str = "log"
  else
    x_str = "linear"
  end if
  if (gro%y_log) then
    y_str = "log"
  else
    y_str = "linear"
  end if
  cmd = "setup (" // x_str // ", " // y_str // ");"
end function graph_options_get_gml_setup

```

Return the labels in GAMELAN form.

```

<Analysis: procedures>+≡
function graph_options_get_gml_x_label (gro) result (cmd)

```



```

type(string_t) :: cmd
type(graph_options_t), intent(in) :: gro
cmd = 'label.bot (< ' // '<' // gro%x_label // '>' // '>', out);'
end function graph_options_get_gml_x_label

function graph_options_get_gml_y_label (gro) result (cmd)
type(string_t) :: cmd
type(graph_options_t), intent(in) :: gro
cmd = 'label.ulft (< ' // '<' // gro%y_label // '>' // '>', out);'
end function graph_options_get_gml_y_label

```

Create an appropriate **graphrange** statement for the given graph options. Where the graph options are not set, use the supplied arguments, if any, otherwise set the undefined value.

*(Analysis: procedures)*+≡

```

function graph_options_get_gml_graphrange &
(gro, x_min, x_max, y_min, y_max) result (cmd)
type(string_t) :: cmd
type(graph_options_t), intent(in) :: gro
real(default), intent(in), optional :: x_min, x_max, y_min, y_max
type(string_t) :: x_min_str, x_max_str, y_min_str, y_max_str
if (gro%x_min_set) then
x_min_str = "#" // real2string (gro%x_min)
else if (present (x_min)) then
x_min_str = "#" // real2string (x_min)
else
x_min_str = "???"
end if
if (gro%x_max_set) then
x_max_str = "#" // real2string (gro%x_max)
else if (present (x_max)) then
x_max_str = "#" // real2string (x_max)
else
x_max_str = "???"
end if
if (gro%y_min_set) then
y_min_str = "#" // real2string (gro%y_min)
else if (present (y_min)) then
y_min_str = "#" // real2string (y_min)
else
y_min_str = "???"
end if
if (gro%y_max_set) then
y_max_str = "#" // real2string (gro%y_max)
else if (present (y_max)) then
y_max_str = "#" // real2string (y_max)
else
y_max_str = "???"
end if
cmd = "graphrange (" // x_min_str // ", " // y_min_str // " ), " &
// "(" // x_max_str // ", " // y_max_str // " );"
end function graph_options_get_gml_graphrange

```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

```

<Analysis: procedures>+=
function graph_options_get_gml_bg_command (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = gro%gmlcode_bg
end function graph_options_get_gml_bg_command

function graph_options_get_gml_fg_command (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = gro%gmlcode_fg
end function graph_options_get_gml_fg_command

```

Append the header for generic data output in ifile format. We print only labels, not graphics parameters.

```

<Analysis: procedures>+=
subroutine graph_options_get_header (pl, header, comment)
  type(graph_options_t), intent(in) :: pl
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  type(string_t) :: c
  if (present (comment)) then
    c = comment
  else
    c = ""
  end if
  call ifile_append (header, &
    c // "ID: " // pl%id)
  call ifile_append (header, &
    c // "title: " // pl%title)
  call ifile_append (header, &
    c // "description: " // pl%description)
  call ifile_append (header, &
    c // "x axis label: " // pl%x_label)
  call ifile_append (header, &
    c // "y axis label: " // pl%y_label)
end subroutine graph_options_get_header

```

### 5.1.3 Drawing options

These options apply to an individual graph element (histogram or plot).

```

<Analysis: public>+=
public :: drawing_options_t

<Analysis: types>+=
type :: drawing_options_t
  type(string_t) :: dataset
  logical :: with_hbars = .false.
  logical :: with_base = .false.
  logical :: piecewise = .false.

```

```

        logical :: fill = .false.
        logical :: draw = .false.
        logical :: err = .false.
        logical :: symbols = .false.
        type(string_t) :: fill_options
        type(string_t) :: draw_options
        type(string_t) :: err_options
        type(string_t) :: symbol
        type(string_t) :: gmlcode_bg
        type(string_t) :: gmlcode_fg
    end type drawing_options_t

```

Init with empty strings and default options, appropriate for either histogram or plot.

*<Analysis: public>+≡*

```

    public :: drawing_options_init_histogram
    public :: drawing_options_init_plot

```

*<Analysis: procedures>+≡*

```

    subroutine drawing_options_init_histogram (dro)
        type(drawing_options_t), intent(out) :: dro
        dro%dataset = "dat"
        dro%with_hbars = .true.
        dro%with_base = .true.
        dro%piecewise = .true.
        dro%fill = .true.
        dro%draw = .true.
        dro%fill_options = "withcolor col.default"
        dro%draw_options = ""
        dro%err_options = ""
        dro%symbol = "fshape(circle scaled 1mm())"
        dro%gmlcode_bg = ""
        dro%gmlcode_fg = ""
    end subroutine drawing_options_init_histogram

    subroutine drawing_options_init_plot (dro)
        type(drawing_options_t), intent(out) :: dro
        dro%dataset = "dat"
        dro%draw = .true.
        dro%fill_options = "withcolor col.default"
        dro%draw_options = ""
        dro%err_options = ""
        dro%symbol = "fshape(circle scaled 1mm())"
        dro%gmlcode_bg = ""
        dro%gmlcode_fg = ""
    end subroutine drawing_options_init_plot

```

Set individual options.

*<Analysis: public>+≡*

```

    public :: drawing_options_set

```

*<Analysis: procedures>+≡*

```

    subroutine drawing_options_set (dro, dataset, &
        with_hbars, with_base, piecewise, fill, draw, err, symbols, &

```

```

        fill_options, draw_options, err_options, symbol, &
        gmlcode_bg, gmlcode_fg)
type(drawing_options_t), intent(inout) :: dro
type(string_t), intent(in), optional :: dataset
logical, intent(in), optional :: with_hbars, with_base, piecewise
logical, intent(in), optional :: fill, draw, err, symbols
type(string_t), intent(in), optional :: fill_options, draw_options
type(string_t), intent(in), optional :: err_options, symbol
type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
if (present (dataset)) dro%dataset = dataset
if (present (with_hbars)) dro%with_hbars = with_hbars
if (present (with_base)) dro%with_base = with_base
if (present (piecewise)) dro%piecewise = piecewise
if (present (fill)) dro%fill = fill
if (present (draw)) dro%draw = draw
if (present (err)) dro%err = err
if (present (symbols)) dro%symbols = symbols
if (present (fill_options)) dro%fill_options = fill_options
if (present (draw_options)) dro%draw_options = draw_options
if (present (err_options)) dro%err_options = err_options
if (present (symbol)) dro%symbol = symbol
if (present (gmlcode_bg)) dro%gmlcode_bg = gmlcode_bg
if (present (gmlcode_fg)) dro%gmlcode_fg = gmlcode_fg
end subroutine drawing_options_set

```

There are separate commands for drawing the curve and for drawing errors. The symbols are applied to the latter. First of all, we may have to compute a baseline:

```

<Analysis: procedures>+≡
function drawing_options_get_calc_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    if (dro%with_base) then
        cmd = "calculate " // dro%dataset // ".base (" // dro%dataset // ") " &
            // "(x, #0);"
    else
        cmd = ""
    end if
end function drawing_options_get_calc_command

```

Return the drawing command.

```

<Analysis: procedures>+≡
function drawing_options_get_draw_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    if (dro%fill) then
        cmd = "fill"
    else if (dro%draw) then
        cmd = "draw"
    else
        cmd = ""
    end if
    if (dro%fill .or. dro%draw) then

```

```

if (dro%piecewise) cmd = cmd // " piecewise"
if (dro%draw .and. dro%with_base) cmd = cmd // " cyclic"
cmd = cmd // " from (" // dro%dataset
if (dro%with_base) then
  if (dro%piecewise) then
    cmd = cmd // ", " // dro%dataset // ".base\" ! "
  else
    cmd = cmd // " ~ " // dro%dataset // ".base\" ! "
  end if
end if
cmd = cmd // ")"
if (dro%fill) then
  cmd = cmd // " " // dro%fill_options
  if (dro%draw) cmd = cmd // " outlined"
end if
if (dro%draw) cmd = cmd // " " // dro%draw_options
cmd = cmd // ";"
end if
end function drawing_options_get_draw_command

```

The error command draws error bars, if any.

*<Analysis: procedures>+≡*

```

function drawing_options_get_err_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%err) then
    cmd = "draw piecewise " &
      // "from (" // dro%dataset // ".err)" &
      // " " // dro%err_options // ";"
  else
    cmd = ""
  end if
end function drawing_options_get_err_command

```

The symbol command draws symbols, if any.

*<Analysis: procedures>+≡*

```

function drawing_options_get_symb_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%symbols) then
    cmd = "phantom" &
      // " from (" // dro%dataset // ")" &
      // " withsymbol (" // dro%symbol // ");"
  else
    cmd = ""
  end if
end function drawing_options_get_symb_command

```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

*<Analysis: procedures>+≡*

```

function drawing_options_get_gml_bg_command (dro) result (cmd)

```

```

    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    cmd = dro%gmlcode_bg
end function drawing_options_get_gml_bg_command

function drawing_options_get_gml_fg_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    cmd = dro%gmlcode_fg
end function drawing_options_get_gml_fg_command

```

#### 5.1.4 Observables

The observable type holds the accumulated observable values and weight sums which are necessary for proper averaging.

*(Analysis: types)*+≡

```

type :: observable_t
    private
    real(default) :: sum_values = 0
    real(default) :: sum_squared_values = 0
    real(default) :: sum_weights = 0
    real(default) :: sum_squared_weights = 0
    integer :: count = 0
    type(string_t) :: obs_label
    type(string_t) :: obs_unit
    type(graph_options_t) :: graph_options
end type observable_t

```

Initialize with defined properties

*(Analysis: procedures)*+≡

```

subroutine observable_init (obs, obs_label, obs_unit, graph_options)
    type(observable_t), intent(out) :: obs
    type(string_t), intent(in), optional :: obs_label, obs_unit
    type(graph_options_t), intent(in), optional :: graph_options
    if (present (obs_label)) then
        obs%obs_label = obs_label
    else
        obs%obs_label = ""
    end if
    if (present (obs_unit)) then
        obs%obs_unit = obs_unit
    else
        obs%obs_unit = ""
    end if
    if (present (graph_options)) then
        obs%graph_options = graph_options
    else
        call graph_options_init (obs%graph_options)
    end if
end subroutine observable_init

```

Reset all numeric entries.

```

<Analysis: procedures>+≡
subroutine observable_clear (obs)
  type(observable_t), intent(inout) :: obs
  obs%sum_values = 0
  obs%sum_squared_values = 0
  obs%sum_weights = 0
  obs%sum_squared_weights = 0
  obs%count = 0
end subroutine observable_clear

```

Record a a value. Always successful for observables.

```

<Analysis: interfaces>≡
interface observable_record_value
  module procedure observable_record_value_unweighted
  module procedure observable_record_value_weighted
end interface

```

```

<Analysis: procedures>+≡
subroutine observable_record_value_unweighted (obs, value, success)
  type(observable_t), intent(inout) :: obs
  real(default), intent(in) :: value
  logical, intent(out), optional :: success
  obs%sum_values = obs%sum_values + value
  obs%sum_squared_values = obs%sum_squared_values + value**2
  obs%sum_weights = obs%sum_weights + 1
  obs%sum_squared_weights = obs%sum_squared_weights + 1
  obs%count = obs%count + 1
  if (present (success)) success = .true.
end subroutine observable_record_value_unweighted

subroutine observable_record_value_weighted (obs, value, weight, success)
  type(observable_t), intent(inout) :: obs
  real(default), intent(in) :: value, weight
  logical, intent(out), optional :: success
  obs%sum_values = obs%sum_values + value * weight
  obs%sum_squared_values = obs%sum_squared_values + value**2 * weight
  obs%sum_weights = obs%sum_weights + abs (weight)
  obs%sum_squared_weights = obs%sum_squared_weights + weight**2
  obs%count = obs%count + 1
  if (present (success)) success = .true.
end subroutine observable_record_value_weighted

```

Here are the statistics formulas:

1. Unweighted case: Given a sample of  $n$  values  $x_i$ , the average is

$$\langle x \rangle = \frac{\sum x_i}{n} \quad (5.1)$$

and the error estimate

$$\Delta x = \sqrt{\frac{1}{n-1} \langle \sum (x_i - \langle x \rangle)^2 \rangle} \quad (5.2)$$

$$= \sqrt{\frac{1}{n-1} \left( \frac{\sum x_i^2}{n} - \frac{(\sum x_i)^2}{n^2} \right)} \quad (5.3)$$

2. Weighted case: Instead of weight 1, each event comes with weight  $w_i$ .

$$\langle x \rangle = \frac{\sum x_i w_i}{\sum w_i} \quad (5.4)$$

and

$$\Delta x = \sqrt{\frac{1}{n-1} \left( \frac{\sum x_i^2 w_i}{\sum w_i} - \frac{(\sum x_i w_i)^2}{(\sum w_i)^2} \right)} \quad (5.5)$$

For  $w_i = 1$ , this specializes to the previous formula.

*<Analysis: procedures>+≡*

```
function observable_get_n_entries (obs) result (n)
  integer :: n
  type(observable_t), intent(in) :: obs
  n = obs%count
end function observable_get_n_entries

function observable_get_average (obs) result (avg)
  real(default) :: avg
  type(observable_t), intent(in) :: obs
  if (obs%sum_weights /= 0) then
    avg = obs%sum_values / obs%sum_weights
  else
    avg = 0
  end if
end function observable_get_average

function observable_get_error (obs) result (err)
  real(default) :: err
  type(observable_t), intent(in) :: obs
  real(default) :: var, n
  if (obs%sum_weights /= 0) then
    select case (obs%count)
    case (0:1)
      err = 0
    case default
      n = obs%count
      var = obs%sum_squared_values / obs%sum_weights &
        - (obs%sum_values / obs%sum_weights) ** 2
      err = sqrt (max (var, 0._default) / (n - 1))
    end select
  else
    err = 0
  end if
end function observable_get_error
```



Write label and/or physical unit to a string.

*<Analysis: procedures>+≡*

```
function observable_get_label (obs, wl, wu) result (string)
  type(string_t) :: string
  type(observable_t), intent(in) :: obs
  logical, intent(in) :: wl, wu
  type(string_t) :: obs_label, obs_unit
  if (wl) then
    if (obs%obs_label /= "") then
      obs_label = obs%obs_label
    else
      obs_label = "\textrm{Observable}"
    end if
  else
    obs_label = ""
  end if
  if (wu) then
    if (obs%obs_unit /= "") then
      if (wl) then
        obs_unit = "\;[" // obs%obs_unit // "]"
      else
        obs_unit = obs%obs_unit
      end if
    else
      obs_unit = ""
    end if
  else
    obs_unit = ""
  end if
  string = obs_label // obs_unit
end function observable_get_label
```

### 5.1.5 Output

*<Analysis: procedures>+≡*

```
subroutine observable_write (obs, unit)
  type(observable_t), intent(in) :: obs
  integer, intent(in), optional :: unit
  real(default) :: avg, err, relerr
  integer :: n
  integer :: u
  u = output_unit (unit); if (u < 0) return
  avg = observable_get_average (obs)
  err = observable_get_error (obs)
  if (avg /= 0) then
    relerr = err / avg
  else
    relerr = 0
  end if
  n = observable_get_n_entries (obs)
  write (u, "(A,ix," // HISTOGRAM_DATA_FORMAT // ")") &
    "average      =", avg
```

```

write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
"error[abs] =", err
write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
"error[rel] =", relerr
write (u, "(A,1x," // HISTOGRAM_INTG_FORMAT // ")") &
"n_entries   =", n
end subroutine observable_write

```

L<sup>A</sup>T<sub>E</sub>X output.

*(Analysis: procedures)*+≡

```

subroutine observable_write_driver (obs, unit, write_heading)
  type(observable_t), intent(in) :: obs
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: write_heading
  real(default) :: avg, err
  integer :: n_digits
  logical :: heading
  integer :: u
  u = output_unit (unit); if (u < 0) return
  heading = .true.; if (present (write_heading)) heading = write_heading
  avg = observable_get_average (obs)
  err = observable_get_error (obs)
  if (avg /= 0 .and. err /= 0) then
    n_digits = max (2, 2 - int (log10 (abs (err / real (avg, default)))))
  else if (avg /= 0) then
    n_digits = 100
  else
    n_digits = 1
  end if
  if (heading) then
    write (u, "(A)")
    if (obs%graph_options%title /= "") then
      write (u, "(A)") "\section{" // char (obs%graph_options%title) &
// "}"
    else
      write (u, "(A)") "\section{Observable}"
    end if
    if (obs%graph_options%description /= "") then
      write (u, "(A)") char (obs%graph_options%description)
      write (u, *)
    end if
    write (u, "(A)") "\begin{flushleft}"
  end if
  write (u, "(A)", advance="no") " $\langle$ " ! $ sign
  write (u, "(A)", advance="no") char (observable_get_label (obs, wl=.true., wu=.false.))
  write (u, "(A)", advance="no") " \rangle = "
  write (u, "(A)", advance="no") char (tex_format (avg, n_digits))
  write (u, "(A)", advance="no") "\pm"
  write (u, "(A)", advance="no") char (tex_format (err, 2))
  write (u, "(A)", advance="no") "\;"
  write (u, "(A)", advance="no") char (observable_get_label (obs, wl=.false., wu=.true.))
  write (u, "(A)") "}"
  write (u, "(A)", advance="no") " \quad [n_{\text{entries}}] = "

```

```

write (u, "(I0)", advance="no")  observable_get_n_entries (obs)
write (u, "(A)")  "]"$          ! $ fool Emacs' noweb mode
if (heading) then
  write (u, "(A)")  "\end{flushleft}"
end if
end subroutine observable_write_driver

```

## 5.1.6 Histograms

### Bins

*<Analysis: types>+≡*

```

type :: bin_t
  private
  real(default) :: midpoint = 0
  real(default) :: width = 0
  real(default) :: sum_weights = 0
  real(default) :: sum_squared_weights = 0
  real(default) :: sum_excess_weights = 0
  integer :: count = 0
end type bin_t

```

*<Analysis: procedures>+≡*

```

subroutine bin_init (bin, midpoint, width)
  type(bin_t), intent(out) :: bin
  real(default), intent(in) :: midpoint, width
  bin%midpoint = midpoint
  bin%width = width
end subroutine bin_init

```

*<Analysis: procedures>+≡*

```

elemental subroutine bin_clear (bin)
  type(bin_t), intent(inout) :: bin
  bin%sum_weights = 0
  bin%sum_squared_weights = 0
  bin%sum_excess_weights = 0
  bin%count = 0
end subroutine bin_clear

```

*<Analysis: procedures>+≡*

```

subroutine bin_record_value (bin, normalize, weight, excess)
  type(bin_t), intent(inout) :: bin
  logical, intent(in) :: normalize
  real(default), intent(in) :: weight
  real(default), intent(in), optional :: excess
  real(default) :: w, e
  if (normalize) then
    if (bin%width /= 0) then
      w = weight / bin%width
      if (present (excess)) e = excess / bin%width
    else
      w = 0

```

```

        if (present (excess)) e = 0
    end if
else
    w = weight
    if (present (excess)) e = excess
    end if
    bin%sum_weights = bin%sum_weights + abs (w)
    bin%sum_squared_weights = bin%sum_squared_weights + w ** 2
    if (present (excess)) &
        bin%sum_excess_weights = bin%sum_excess_weights + abs (e)
    bin%count = bin%count + 1
end subroutine bin_record_value

```

*(Analysis: procedures)*+≡

```

function bin_get_midpoint (bin) result (x)
    real(default) :: x
    type(bin_t), intent(in) :: bin
    x = bin%midpoint
end function bin_get_midpoint

function bin_get_width (bin) result (w)
    real(default) :: w
    type(bin_t), intent(in) :: bin
    w = bin%width
end function bin_get_width

function bin_get_n_entries (bin) result (n)
    integer :: n
    type(bin_t), intent(in) :: bin
    n = bin%count
end function bin_get_n_entries

function bin_get_sum (bin) result (s)
    real(default) :: s
    type(bin_t), intent(in) :: bin
    s = bin%sum_weights
end function bin_get_sum

function bin_get_error (bin) result (err)
    real(default) :: err
    type(bin_t), intent(in) :: bin
    err = sqrt (bin%sum_squared_weights)
end function bin_get_error

function bin_get_excess (bin) result (excess)
    real(default) :: excess
    type(bin_t), intent(in) :: bin
    excess = bin%sum_excess_weights
end function bin_get_excess

```

*(Analysis: procedures)*+≡

```

subroutine bin_write_header (unit)
    integer, intent(in), optional :: unit

```

```

character(120) :: buffer
integer :: u
u = output_unit (unit); if (u < 0) return
write (buffer, "(A,5(1x," // HISTOGRAM_HEAD_FORMAT // "))" &
      "#", "bin midpoint", "value      ", "error      ", &
      "n_entries ", "excess      ")
write (u, "(A)") trim (buffer)
end subroutine bin_write_header

subroutine bin_write (bin, unit)
type(bin_t), intent(in) :: bin
integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit); if (u < 0) return
write (u, "(1x,3(1x," // HISTOGRAM_DATA_FORMAT // ")," &
      // HISTOGRAM_INTG_FORMAT // "," &
      // HISTOGRAM_DATA_FORMAT // ")") &
      bin_get_midpoint (bin), &
      bin_get_sum (bin), &
      bin_get_error (bin), &
      bin_get_n_entries (bin), &
      bin_get_excess (bin)
end subroutine bin_write

```

## Histograms

*<Analysis: types>+≡*

```

type :: histogram_t
private
real(default) :: lower_bound = 0
real(default) :: upper_bound = 0
real(default) :: width = 0
integer :: n_bins = 0
logical :: normalize_bins = .false.
type(observable_t) :: obs
type(observable_t) :: obs_within_bounds
type(bin_t) :: underflow
type(bin_t), dimension(:), allocatable :: bin
type(bin_t) :: overflow
type(graph_options_t) :: graph_options
type(drawing_options_t) :: drawing_options
end type histogram_t

```

## Initializer/finalizer

Initialize a histogram. We may provide either the bin width or the number of bins. A finalizer is not needed, since the histogram contains no pointer (sub)components.

*<Analysis: interfaces>+≡*

```

interface histogram_init
module procedure histogram_init_n_bins

```

```

    module procedure histogram_init_bin_width
end interface

```

*(Analysis: procedures)*+≡

```

subroutine histogram_init_n_bins (h, id, &
    lower_bound, upper_bound, n_bins, normalize_bins, &
    obs_label, obs_unit, graph_options, drawing_options)
type(histogram_t), intent(out) :: h
type(string_t), intent(in) :: id
real(default), intent(in) :: lower_bound, upper_bound
integer, intent(in) :: n_bins
logical, intent(in) :: normalize_bins
type(string_t), intent(in), optional :: obs_label, obs_unit
type(graph_options_t), intent(in), optional :: graph_options
type(drawing_options_t), intent(in), optional :: drawing_options
real(default) :: bin_width
integer :: i
call observable_init (h%obs_within_bounds, obs_label, obs_unit)
call observable_init (h%obs, obs_label, obs_unit)
h%lower_bound = lower_bound
h%upper_bound = upper_bound
h%n_bins = max (n_bins, 1)
h%width = h%upper_bound - h%lower_bound
h%normalize_bins = normalize_bins
bin_width = h%width / h%n_bins
allocate (h%bin (h%n_bins))
call bin_init (h%underflow, h%lower_bound, 0._default)
do i = 1, h%n_bins
    call bin_init (h%bin(i), &
        h%lower_bound - bin_width/2 + i * bin_width, bin_width)
end do
call bin_init (h%overflow, h%upper_bound, 0._default)
if (present (graph_options)) then
    h%graph_options = graph_options
else
    call graph_options_init (h%graph_options)
end if
call graph_options_set (h%graph_options, id = id)
if (present (drawing_options)) then
    h%drawing_options = drawing_options
else
    call drawing_options_init_histogram (h%drawing_options)
end if
end subroutine histogram_init_n_bins

subroutine histogram_init_bin_width (h, id, &
    lower_bound, upper_bound, bin_width, normalize_bins, &
    obs_label, obs_unit, graph_options, drawing_options)
type(histogram_t), intent(out) :: h
type(string_t), intent(in) :: id
real(default), intent(in) :: lower_bound, upper_bound, bin_width
logical, intent(in) :: normalize_bins
type(string_t), intent(in), optional :: obs_label, obs_unit
type(graph_options_t), intent(in), optional :: graph_options

```

```

type(drawing_options_t), intent(in), optional :: drawing_options
integer :: n_bins
if (bin_width /= 0) then
    n_bins = nint ((upper_bound - lower_bound) / bin_width)
else
    n_bins = 1
end if
call histogram_init_n_bins (h, id, &
    lower_bound, upper_bound, n_bins, normalize_bins, &
    obs_label, obs_unit, graph_options, drawing_options)
end subroutine histogram_init_bin_width

```

Initialize a histogram by copying another one.

Since `h` has no pointer (sub)components, intrinsic assignment is sufficient. Optionally, we replace the drawing options.

*<Analysis: procedures>+≡*

```

subroutine histogram_init_histogram (h, h_in, drawing_options)
    type(histogram_t), intent(out) :: h
    type(histogram_t), intent(in) :: h_in
    type(drawing_options_t), intent(in), optional :: drawing_options
    h = h_in
    if (present (drawing_options)) then
        h%drawing_options = drawing_options
    end if
end subroutine histogram_init_histogram

```

## Fill histograms

Clear the histogram contents, but do not modify the structure.

*<Analysis: procedures>+≡*

```

subroutine histogram_clear (h)
    type(histogram_t), intent(inout) :: h
    call observable_clear (h%obs)
    call observable_clear (h%obs_within_bounds)
    call bin_clear (h%underflow)
    if (allocated (h%bin)) call bin_clear (h%bin)
    call bin_clear (h%overflow)
end subroutine histogram_clear

```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow. Optionally, we may provide an excess weight that could be returned by the unweighting procedure.

*<Analysis: procedures>+≡*

```

subroutine histogram_record_value_unweighted (h, value, excess, success)
    type(histogram_t), intent(inout) :: h
    real(default), intent(in) :: value
    real(default), intent(in), optional :: excess
    logical, intent(out), optional :: success
    integer :: i_bin
    call observable_record_value (h%obs, value)
    if (h%width /= 0) then

```

```

        i_bin = floor (((value - h%lower_bound) / h%width) * h%n_bins) + 1
    else
        i_bin = 0
    end if
    if (i_bin <= 0) then
        call bin_record_value (h%underflow, .false., 1._default, excess)
        if (present (success)) success = .false.
    else if (i_bin <= h%n_bins) then
        call observable_record_value (h%obs_within_bounds, value)
        call bin_record_value &
            (h%bin(i_bin), h%normalize_bins, 1._default, excess)
        if (present (success)) success = .true.
    else
        call bin_record_value (h%overflow, .false., 1._default, excess)
        if (present (success)) success = .false.
    end if
end subroutine histogram_record_value_unweighted

```

Weighted events: analogous, but no excess weight.

*<Analysis: procedures>+≡*

```

subroutine histogram_record_value_weighted (h, value, weight, success)
    type(histogram_t), intent(inout) :: h
    real(default), intent(in) :: value, weight
    logical, intent(out), optional :: success
    integer :: i_bin
    call observable_record_value (h%obs, value, weight)
    if (h%width /= 0) then
        i_bin = floor (((value - h%lower_bound) / h%width) * h%n_bins) + 1
    else
        i_bin = 0
    end if
    if (i_bin <= 0) then
        call bin_record_value (h%underflow, .false., weight)
        if (present (success)) success = .false.
    else if (i_bin <= h%n_bins) then
        call observable_record_value (h%obs_within_bounds, value, weight)
        call bin_record_value (h%bin(i_bin), h%normalize_bins, weight)
        if (present (success)) success = .true.
    else
        call bin_record_value (h%overflow, .false., weight)
        if (present (success)) success = .false.
    end if
end subroutine histogram_record_value_weighted

```

## Access contents

Inherited from the observable component (all-over average etc.)

*<Analysis: procedures>+≡*

```

function histogram_get_n_entries (h) result (n)
    integer :: n
    type(histogram_t), intent(in) :: h
    n = observable_get_n_entries (h%obs)

```



```

end function histogram_get_n_entries

function histogram_get_average (h) result (avg)
  real(default) :: avg
  type(histogram_t), intent(in) :: h
  avg = observable_get_average (h%obs)
end function histogram_get_average

function histogram_get_error (h) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  err = observable_get_error (h%obs)
end function histogram_get_error

```

Analogous, but applied only to events within bounds.

*<Analysis: procedures>+≡*

```

function histogram_get_n_entries_within_bounds (h) result (n)
  integer :: n
  type(histogram_t), intent(in) :: h
  n = observable_get_n_entries (h%obs_within_bounds)
end function histogram_get_n_entries_within_bounds

function histogram_get_average_within_bounds (h) result (avg)
  real(default) :: avg
  type(histogram_t), intent(in) :: h
  avg = observable_get_average (h%obs_within_bounds)
end function histogram_get_average_within_bounds

function histogram_get_error_within_bounds (h) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  err = observable_get_error (h%obs_within_bounds)
end function histogram_get_error_within_bounds

```

Get the number of bins

*<Analysis: procedures>+≡*

```

function histogram_get_n_bins (h) result (n)
  type(histogram_t), intent(in) :: h
  integer :: n
  n = h%n_bins
end function histogram_get_n_bins

```

Check bins. If the index is zero or above the limit, return the results for underflow or overflow, respectively.

*<Analysis: procedures>+≡*

```

function histogram_get_n_entries_for_bin (h, i) result (n)
  integer :: n
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
    n = bin_get_n_entries (h%underflow)
  else if (i <= h%n_bins) then

```

```

        n = bin_get_n_entries (h%bin(i))
    else
        n = bin_get_n_entries (h%overflow)
    end if
end function histogram_get_n_entries_for_bin

function histogram_get_sum_for_bin (h, i) result (avg)
    real(default) :: avg
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
        avg = bin_get_sum (h%underflow)
    else if (i <= h%n_bins) then
        avg = bin_get_sum (h%bin(i))
    else
        avg = bin_get_sum (h%overflow)
    end if
end function histogram_get_sum_for_bin

function histogram_get_error_for_bin (h, i) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
        err = bin_get_error (h%underflow)
    else if (i <= h%n_bins) then
        err = bin_get_error (h%bin(i))
    else
        err = bin_get_error (h%overflow)
    end if
end function histogram_get_error_for_bin

function histogram_get_excess_for_bin (h, i) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
        err = bin_get_excess (h%underflow)
    else if (i <= h%n_bins) then
        err = bin_get_excess (h%bin(i))
    else
        err = bin_get_excess (h%overflow)
    end if
end function histogram_get_excess_for_bin

```

Return a pointer to the graph options.

*<Analysis: procedures>+≡*

```

function histogram_get_graph_options_ptr (h) result (ptr)
    type(graph_options_t), pointer :: ptr
    type(histogram_t), intent(in), target :: h
    ptr => h%graph_options
end function histogram_get_graph_options_ptr

```

Return a pointer to the drawing options.

*<Analysis: procedures>+≡*

```
function histogram_get_drawing_options_ptr (h) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(histogram_t), intent(in), target :: h
  ptr => h%drawing_options
end function histogram_get_drawing_options_ptr
```

## Output

*<Analysis: procedures>+≡*

```
subroutine histogram_write (h, unit)
  type(histogram_t), intent(in) :: h
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call bin_write_header (u)
  if (allocated (h%bin)) then
    do i = 1, h%n_bins
      call bin_write (h%bin(i), u)
    end do
  end if
  write (u, *)
  write (u, "(A,1x,A)" "#", "Underflow:")
  call bin_write (h%underflow, u)
  write (u, *)
  write (u, "(A,1x,A)" "#", "Overflow:")
  call bin_write (h%overflow, u)
  write (u, *)
  write (u, "(A,1x,A)" "#", "Summary: data within bounds")
  call observable_write (h%obs_within_bounds, u)
  write (u, *)
  write (u, "(A,1x,A)" "#", "Summary: all data")
  call observable_write (h%obs, u)
  write (u, *)
end subroutine histogram_write
```

Write the GAMELAN reader for histogram contents.

*<Analysis: procedures>+≡*

```
subroutine histogram_write_gml_reader (h, filename, unit)
  type(histogram_t), intent(in) :: h
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(2x,A)" 'fromfile ' // char (filename) // '":')
  write (u, "(4x,A)" 'key "# Histogram:;')
  write (u, "(4x,A)" 'dx := #' &
    // real2char (h%width / h%n_bins / 2) // ',')
  write (u, "(4x,A)" 'for i withinblock:')
  write (u, "(6x,A)" 'get x, y, y.d, y.n, y.e;')
  if (h%drawing_options%with_hbars) then
```

```

        write (u, "(6x,A)") 'plot (' // char (h%drawing_options%dataset) &
            // ')' (x,y) hbar dx;'
    else
        write (u, "(6x,A)") 'plot (' // char (h%drawing_options%dataset) &
            // ')' (x,y);'
    end if
    if (h%drawing_options%err) then
        write (u, "(6x,A)") 'plot (' // char (h%drawing_options%dataset) &
            // '.err) ' &
            // '(x,y) vbar y.d;'
    end if
!   write (u, "(6x,A)") 'if show_excess: ' // &
!       & 'plot(dat.e)(x, y plus y.e) hbar dx; fi'
    write (u, "(4x,A)") 'endfor'
    write (u, "(2x,A)") 'endfrom'
end subroutine histogram_write_gml_reader

```

L<sup>A</sup>T<sub>E</sub>X and GAMELAN output.

*(Analysis: procedures)*+≡

```

subroutine histogram_write_gml_driver (h, filename, unit)
    type(histogram_t), intent(in) :: h
    type(string_t), intent(in) :: filename
    integer, intent(in), optional :: unit
    type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call graph_options_write_tex_header (h%graph_options, unit)
    write (u, "(2x,A)") char (graph_options_get_gml_setup (h%graph_options))
    write (u, "(2x,A)") char (graph_options_get_gml_graphrange &
        (h%graph_options, x_min=h%lower_bound, x_max=h%upper_bound))
    call histogram_write_gml_reader (h, filename, unit)
    calc_cmd = drawing_options_get_calc_command (h%drawing_options)
    if (calc_cmd /= "") write (u, "(2x,A)") char (calc_cmd)
    bg_cmd = drawing_options_get_gml_bg_command (h%drawing_options)
    if (bg_cmd /= "") write (u, "(2x,A)") char (bg_cmd)
    draw_cmd = drawing_options_get_draw_command (h%drawing_options)
    if (draw_cmd /= "") write (u, "(2x,A)") char (draw_cmd)
    err_cmd = drawing_options_get_err_command (h%drawing_options)
    if (err_cmd /= "") write (u, "(2x,A)") char (err_cmd)
    symb_cmd = drawing_options_get_symb_command (h%drawing_options)
    if (symb_cmd /= "") write (u, "(2x,A)") char (symb_cmd)
    fg_cmd = drawing_options_get_gml_fg_command (h%drawing_options)
    if (fg_cmd /= "") write (u, "(2x,A)") char (fg_cmd)
    write (u, "(2x,A)") char (graph_options_get_gml_x_label (h%graph_options))
    write (u, "(2x,A)") char (graph_options_get_gml_y_label (h%graph_options))
    call graph_options_write_tex_footer (h%graph_options, unit)
    write (u, "(A)") "\vspace*{2\baselineskip}"
    write (u, "(A)") "\begin{flushleft}"
    write (u, "(A)") "\textbf{Data within bounds:} \\"
    call observable_write_driver (h%obs_within_bounds, unit, &
        write_heading=.false.)
    write (u, "(A)") "\\[0.5\baselineskip]"
    write (u, "(A)") "\textbf{All data:} \\"
    call observable_write_driver (h%obs, unit, write_heading=.false.)

```

```

        write (u, "(A)") "\end{flushleft}"
    end subroutine histogram_write_gml_driver

```

Return the header for generic data output as an ifile.

```

<Analysis: procedures>+≡
subroutine histogram_get_header (h, header, comment)
    type(histogram_t), intent(in) :: h
    type(ifile_t), intent(inout) :: header
    type(string_t), intent(in), optional :: comment
    type(string_t) :: c
    if (present (comment)) then
        c = comment
    else
        c = ""
    end if
    call ifile_append (header, c // "WHIZARD histogram data")
    call graph_options_get_header (h%graph_options, header, comment)
    call ifile_append (header, &
        c // "range: " // real2string (h%lower_bound) &
        // " - " // real2string (h%upper_bound))
    call ifile_append (header, &
        c // "counts total: " &
        // int2char (histogram_get_n_entries_within_bounds (h)))
    call ifile_append (header, &
        c // "total average: " &
        // real2string (histogram_get_average_within_bounds (h)) // " +- " &
        // real2string (histogram_get_error_within_bounds (h)))
end subroutine histogram_get_header

```

## 5.1.7 Plots

### Points

```

<Analysis: types>+≡
type :: point_t
    private
    real(default) :: x = 0
    real(default) :: y = 0
    real(default) :: yerr = 0
    real(default) :: xerr = 0
    type(point_t), pointer :: next => null ()
end type point_t

<Analysis: interfaces>+≡
interface point_init
    module procedure point_init_contents
    module procedure point_init_point
end interface

<Analysis: procedures>+≡
subroutine point_init_contents (point, x, y, yerr, xerr)
    type(point_t), intent(out) :: point
    real(default), intent(in) :: x, y

```

```

    real(default), intent(in), optional :: yerr, xerr
    point%x = x
    point%y = y
    if (present (yerr)) point%yerr = yerr
    if (present (xerr)) point%xerr = xerr
end subroutine point_init_contents

subroutine point_init_point (point, point_in)
    type(point_t), intent(out) :: point
    type(point_t), intent(in) :: point_in
    point%x = point_in%x
    point%y = point_in%y
    point%yerr = point_in%yerr
    point%xerr = point_in%xerr
end subroutine point_init_point

```

*<Analysis: procedures>+≡*

```

function point_get_x (point) result (x)
    real(default) :: x
    type(point_t), intent(in) :: point
    x = point%x
end function point_get_x

function point_get_y (point) result (y)
    real(default) :: y
    type(point_t), intent(in) :: point
    y = point%y
end function point_get_y

function point_get_xerr (point) result (xerr)
    real(default) :: xerr
    type(point_t), intent(in) :: point
    xerr = point%xerr
end function point_get_xerr

function point_get_yerr (point) result (yerr)
    real(default) :: yerr
    type(point_t), intent(in) :: point
    yerr = point%yerr
end function point_get_yerr

```

*<Analysis: procedures>+≡*

```

subroutine point_write_header (unit)
    integer, intent(in) :: unit
    character(120) :: buffer
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (buffer, "(A,4(1x," // HISTOGRAM_HEAD_FORMAT // "))" &
        & "#", "x", "y", "yerr", "xerr")
    write (u, "(A)") trim (buffer)
end subroutine point_write_header

subroutine point_write (point, unit)

```

```

type(point_t), intent(in) :: point
integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit); if (u < 0) return
write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // "))" &
      point_get_x (point), &
      point_get_y (point), &
      point_get_yerr (point), &
      point_get_xerr (point)
end subroutine point_write

```

## Plots

*<Analysis: types>+≡*

```

type :: plot_t
private
type(point_t), pointer :: first => null ()
type(point_t), pointer :: last => null ()
integer :: count = 0
type(graph_options_t) :: graph_options
type(drawing_options_t) :: drawing_options
end type plot_t

```

## Initializer/finalizer

Initialize a plot. We provide the lower and upper bound in the  $x$  direction.

*<Analysis: interfaces>+≡*

```

interface plot_init
  module procedure plot_init_empty
  module procedure plot_init_plot
end interface

```

*<Analysis: procedures>+≡*

```

subroutine plot_init_empty (p, id, graph_options, drawing_options)
  type(plot_t), intent(out) :: p
  type(string_t), intent(in) :: id
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  if (present (graph_options)) then
    p%graph_options = graph_options
  else
    call graph_options_init (p%graph_options)
  end if
  call graph_options_set (p%graph_options, id = id)
  if (present (drawing_options)) then
    p%drawing_options = drawing_options
  else
    call drawing_options_init_plot (p%drawing_options)
  end if
end subroutine plot_init_empty

```

Initialize a plot by copying another one, optionally merging in a new set of drawing options.

Since `p` has pointer (sub)components, we have to explicitly deep-copy the original.

*<Analysis: procedures>+≡*

```
subroutine plot_init_plot (p, p_in, drawing_options)
  type(plot_t), intent(out) :: p
  type(plot_t), intent(in) :: p_in
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(point_t), pointer :: current, new
  current => p_in%first
  do while (associated (current))
    allocate (new)
    call point_init (new, current)
    if (associated (p%last)) then
      p%last%next => new
    else
      p%first => new
    end if
    p%last => new
    current => current%next
  end do
  p%count = p_in%count
  p%graph_options = p_in%graph_options
  if (present (drawing_options)) then
    p%drawing_options = drawing_options
  else
    p%drawing_options = p_in%drawing_options
  end if
end subroutine plot_init_plot
```

Finalize the plot by deallocating the list of points.

*<Analysis: procedures>+≡*

```
subroutine plot_final (plot)
  type(plot_t), intent(inout) :: plot
  type(point_t), pointer :: current
  do while (associated (plot%first))
    current => plot%first
    plot%first => current%next
    deallocate (current)
  end do
  plot%last => null ()
end subroutine plot_final
```

## Fill plots

Clear the plot contents, but do not modify the structure.

*<Analysis: procedures>+≡*

```
subroutine plot_clear (plot)
  type(plot_t), intent(inout) :: plot
  plot%count = 0
```



```

        call plot_final (plot)
    end subroutine plot_clear

```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow.

*<Analysis: procedures>+≡*

```

subroutine plot_record_value (plot, x, y, yerr, xerr, success)
    type(plot_t), intent(inout) :: plot
    real(default), intent(in) :: x, y
    real(default), intent(in), optional :: yerr, xerr
    logical, intent(out), optional :: success
    type(point_t), pointer :: point
    plot%count = plot%count + 1
    allocate (point)
    call point_init (point, x, y, yerr, xerr)
    if (associated (plot%first)) then
        plot%last%next => point
    else
        plot%first => point
    end if
    plot%last => point
    if (present (success)) success = .true.
end subroutine plot_record_value

```

## Access contents

The number of points.

*<Analysis: procedures>+≡*

```

function plot_get_n_entries (plot) result (n)
    integer :: n
    type(plot_t), intent(in) :: plot
    n = plot%count
end function plot_get_n_entries

```

Return a pointer to the graph options.

*<Analysis: procedures>+≡*

```

function plot_get_graph_options_ptr (p) result (ptr)
    type(graph_options_t), pointer :: ptr
    type(plot_t), intent(in), target :: p
    ptr => p%graph_options
end function plot_get_graph_options_ptr

```

Return a pointer to the drawing options.

*<Analysis: procedures>+≡*

```

function plot_get_drawing_options_ptr (p) result (ptr)
    type(drawing_options_t), pointer :: ptr
    type(plot_t), intent(in), target :: p
    ptr => p%drawing_options
end function plot_get_drawing_options_ptr

```

## Output

This output format is used by the GAMELAN driver below.

*<Analysis: procedures>+≡*

```
subroutine plot_write (plot, unit)
  type(plot_t), intent(in) :: plot
  integer, intent(in), optional :: unit
  type(point_t), pointer :: point
  integer :: u
  u = output_unit (unit); if (u < 0) return
  call point_write_header (u)
  point => plot%first
  do while (associated (point))
    call point_write (point, unit)
    point => point%next
  end do
  write (u, *)
  write (u, "(A,1x,A)" ) "#", "Summary:"
  write (u, "(A," // HISTOGRAM_INTG_FORMAT // ")") &
    "n_entries = ", plot_get_n_entries (plot)
  write (u, *)
end subroutine plot_write
```

Write the GAMELAN reader for plot contents.

*<Analysis: procedures>+≡*

```
subroutine plot_write_gml_reader (p, filename, unit)
  type(plot_t), intent(in) :: p
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(2x,A)" ) 'fromfile "' // char (filename) // '":'
  write (u, "(4x,A)" ) 'key "# Plot:";'
  write (u, "(4x,A)" ) 'for i withinblock:'
  write (u, "(6x,A)" ) 'get x, y, y.err, x.err;'
  write (u, "(6x,A)" ) 'plot (' // char (p%drawing_options%dataset) &
    // ')' (x,y);'
  if (p%drawing_options%err) then
    write (u, "(6x,A)" ) 'plot (' // char (p%drawing_options%dataset) &
      // '.err) (x,y) vbar y.err hbar x.err;'
  end if
  write (u, "(4x,A)" ) 'endfor'
  write (u, "(2x,A)" ) 'endfrom'
end subroutine plot_write_gml_reader
```

L<sup>A</sup>T<sub>E</sub>X and GAMELAN output. Analogous to histogram output.

*<Analysis: procedures>+≡*

```
subroutine plot_write_gml_driver (p, filename, unit)
  type(plot_t), intent(in) :: p
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
  integer :: u
```

```

u = output_unit (unit); if (u < 0) return
call graph_options_write_tex_header (p%graph_options, unit)
write (u, "(2x,A)") &
    char (graph_options_get_gml_setup (p%graph_options))
write (u, "(2x,A)") &
    char (graph_options_get_gml_graphrange (p%graph_options))
call plot_write_gml_reader (p, filename, unit)
calc_cmd = drawing_options_get_calc_command (p%drawing_options)
if (calc_cmd /= "") write (u, "(2x,A)") char (calc_cmd)
bg_cmd = drawing_options_get_gml_bg_command (p%drawing_options)
if (bg_cmd /= "") write (u, "(2x,A)") char (bg_cmd)
draw_cmd = drawing_options_get_draw_command (p%drawing_options)
if (draw_cmd /= "") write (u, "(2x,A)") char (draw_cmd)
err_cmd = drawing_options_get_err_command (p%drawing_options)
if (err_cmd /= "") write (u, "(2x,A)") char (err_cmd)
symb_cmd = drawing_options_get_symb_command (p%drawing_options)
if (symb_cmd /= "") write (u, "(2x,A)") char (symb_cmd)
fg_cmd = drawing_options_get_gml_fg_command (p%drawing_options)
if (fg_cmd /= "") write (u, "(2x,A)") char (fg_cmd)
write (u, "(2x,A)") char (graph_options_get_gml_x_label (p%graph_options))
write (u, "(2x,A)") char (graph_options_get_gml_y_label (p%graph_options))
call graph_options_write_tex_footer (p%graph_options, unit)
end subroutine plot_write_gml_driver

```

Append header for generic data output in ifile format.

*(Analysis: procedures)*+≡

```

subroutine plot_get_header (plot, header, comment)
    type(plot_t), intent(in) :: plot
    type(ifile_t), intent(inout) :: header
    type(string_t), intent(in), optional :: comment
    type(string_t) :: c
    if (present (comment)) then
        c = comment
    else
        c = ""
    end if
    call ifile_append (header, c // "WHIZARD plot data")
    call graph_options_get_header (plot%graph_options, header, comment)
    call ifile_append (header, &
        c // "number of points: " &
        // int2char (plot_get_n_entries (plot)))
end subroutine plot_get_header

```

### 5.1.8 Graphs

A graph is a container for several graph elements. Each graph element is either a plot or a histogram. There is an appropriate base type below (the `analysis_object_t`), but to avoid recursion, we define a separate base type here. Note that there is no actual recursion: a graph is an analysis object, but a graph cannot contain graphs.

(If we could use type extension, the implementation would be much more transparent.)

## Graph elements

Graph elements cannot be filled by the **record** command directly. The contents are always copied from elementary histograms or plots.

*(Analysis: types)+≡*

```
type :: graph_element_t
private
integer :: type = AN_UNDEFINED
type(histogram_t), pointer :: h => null ()
type(plot_t), pointer :: p => null ()
end type graph_element_t
```

*(Analysis: procedures)+≡*

```
subroutine graph_element_final (el)
type(graph_element_t), intent(inout) :: el
select case (el%type)
case (AN_HISTOGRAM)
deallocate (el%h)
case (AN_PLOT)
call plot_final (el%p)
deallocate (el%p)
end select
el%type = AN_UNDEFINED
end subroutine graph_element_final
```

Return the number of entries in the graph element:

*(Analysis: procedures)+≡*

```
function graph_element_get_n_entries (el) result (n)
integer :: n
type(graph_element_t), intent(in) :: el
select case (el%type)
case (AN_HISTOGRAM); n = histogram_get_n_entries (el%h)
case (AN_PLOT);      n = plot_get_n_entries (el%p)
case default;        n = 0
end select
end function graph_element_get_n_entries
```

Return a pointer to the graph / drawing options.

*(Analysis: procedures)+≡*

```
function graph_element_get_graph_options_ptr (el) result (ptr)
type(graph_options_t), pointer :: ptr
type(graph_element_t), intent(in) :: el
select case (el%type)
case (AN_HISTOGRAM); ptr => histogram_get_graph_options_ptr (el%h)
case (AN_PLOT);      ptr => plot_get_graph_options_ptr (el%p)
case default;        ptr => null ()
end select
end function graph_element_get_graph_options_ptr

function graph_element_get_drawing_options_ptr (el) result (ptr)
type(drawing_options_t), pointer :: ptr
type(graph_element_t), intent(in) :: el
```

```

select case (el%type)
case (AN_HISTOGRAM); ptr => histogram_get_drawing_options_ptr (el%h)
case (AN_PLOT);      ptr => plot_get_drawing_options_ptr (el%p)
case default;        ptr => null ()
end select
end function graph_element_get_drawing_options_ptr

```

Output, simple wrapper for the plot/histogram writer.

*<Analysis: procedures>+≡*

```

subroutine graph_element_write (el, unit)
  type(graph_element_t), intent(in) :: el
  integer, intent(in), optional :: unit
  type(graph_options_t), pointer :: gro
  type(string_t) :: id
  integer :: u
  u = output_unit (unit); if (u < 0) return
  gro => graph_element_get_graph_options_ptr (el)
  id = graph_options_get_id (gro)
  write (u, "(A,A)" ' #', repeat ("-", 78))
  select case (el%type)
  case (AN_HISTOGRAM)
    write (u, "(A)", advance="no")  "# Histogram: "
    write (u, "(1x,A)" char (id)
    call histogram_write (el%h, unit)
  case (AN_PLOT)
    write (u, "(A)", advance="no")  "# Plot: "
    write (u, "(1x,A)" char (id)
    call plot_write (el%p, unit)
  end select
end subroutine graph_element_write

```

*<Analysis: procedures>+≡*

```

subroutine graph_element_write_gml_reader (el, filename, unit)
  type(graph_element_t), intent(in) :: el
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  select case (el%type)
  case (AN_HISTOGRAM); call histogram_write_gml_reader (el%h, filename, unit)
  case (AN_PLOT);      call plot_write_gml_reader (el%p, filename, unit)
  end select
end subroutine graph_element_write_gml_reader

```

## The graph type

The actual graph type contains its own `graph_options`, which override the individual settings. The `drawing_options` are set in the graph elements. This distinction motivates the separation of the two types.

*<Analysis: types>+≡*

```

type :: graph_t
  private
  type(graph_element_t), dimension(:), allocatable :: el

```

```

    type(graph_options_t) :: graph_options
end type graph_t

```

### Initializer/finalizer

The graph is created with a definite number of elements. The elements are filled one by one, optionally with modified drawing options.

*<Analysis: procedures>+≡*

```

subroutine graph_init (g, id, n_elements, graph_options)
    type(graph_t), intent(out) :: g
    type(string_t), intent(in) :: id
    integer, intent(in) :: n_elements
    type(graph_options_t), intent(in), optional :: graph_options
    allocate (g%el (n_elements))
    if (present (graph_options)) then
        g%graph_options = graph_options
    else
        call graph_options_init (g%graph_options)
    end if
    call graph_options_set (g%graph_options, id = id)
end subroutine graph_init

```

*<Analysis: procedures>+≡*

```

subroutine graph_insert_histogram (g, i, h, drawing_options)
    type(graph_t), intent(inout), target :: g
    integer, intent(in) :: i
    type(histogram_t), intent(in) :: h
    type(drawing_options_t), intent(in), optional :: drawing_options
    type(graph_options_t), pointer :: gro
    type(drawing_options_t), pointer :: dro
    type(string_t) :: id
    g%el(i)%type = AN_HISTOGRAM
    allocate (g%el(i)%h)
    call histogram_init_histogram (g%el(i)%h, h, drawing_options)
    gro => histogram_get_graph_options_ptr (g%el(i)%h)
    dro => histogram_get_drawing_options_ptr (g%el(i)%h)
    id = graph_options_get_id (gro)
    call drawing_options_set (dro, dataset = "dat." // id)
end subroutine graph_insert_histogram

```

*<Analysis: procedures>+≡*

```

subroutine graph_insert_plot (g, i, p, drawing_options)
    type(graph_t), intent(inout) :: g
    integer, intent(in) :: i
    type(plot_t), intent(in) :: p
    type(drawing_options_t), intent(in), optional :: drawing_options
    type(graph_options_t), pointer :: gro
    type(drawing_options_t), pointer :: dro
    type(string_t) :: id
    g%el(i)%type = AN_PLOT
    allocate (g%el(i)%p)
    call plot_init_plot (g%el(i)%p, p, drawing_options)

```

```

        gro => plot_get_graph_options_ptr (g%el(i)%p)
        dro => plot_get_drawing_options_ptr (g%el(i)%p)
        id = graph_options_get_id (gro)
        call drawing_options_set (dro, dataset = "dat." // id)
    end subroutine graph_insert_plot

```

Finalizer.

```

<Analysis: procedures>+≡
subroutine graph_final (g)
    type(graph_t), intent(inout) :: g
    integer :: i
    do i = 1, size (g%el)
        call graph_element_final (g%el(i))
    end do
    deallocate (g%el)
end subroutine graph_final

```

## Access contents

The number of elements.

```

<Analysis: procedures>+≡
function graph_get_n_elements (graph) result (n)
    integer :: n
    type(graph_t), intent(in) :: graph
    n = size (graph%el)
end function graph_get_n_elements

```

Retrieve a pointer to the drawing options of an element, so they can be modified. (The `target` attribute is not actually needed because the components are pointers.)

```

<Analysis: procedures>+≡
function graph_get_drawing_options_ptr (g, i) result (ptr)
    type(drawing_options_t), pointer :: ptr
    type(graph_t), intent(in), target :: g
    integer, intent(in) :: i
    ptr => graph_element_get_drawing_options_ptr (g%el(i))
end function graph_get_drawing_options_ptr

```

## Output

The default output format just writes histogram and plot data.

```

<Analysis: procedures>+≡
subroutine graph_write (graph, unit)
    type(graph_t), intent(in) :: graph
    integer, intent(in), optional :: unit
    integer :: i
    do i = 1, size (graph%el)
        call graph_element_write (graph%el(i), unit)
    end do

```

```
end subroutine graph_write
```

The GAMELAN driver is not a simple wrapper, but it writes the plot/histogram contents embedded the complete graph. First, data are read in, global background commands next, then individual elements, then global foreground commands.

*<Analysis: procedures>+≡*

```
subroutine graph_write_gml_driver (g, filename, unit)
  type(graph_t), intent(in) :: g
  type(string_t), intent(in) :: filename
  type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
  integer, intent(in), optional :: unit
  type(drawing_options_t), pointer :: dro
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call graph_options_write_tex_header (g%graph_options, unit)
  write (u, "(2x,A)") &
    char (graph_options_get_gml_setup (g%graph_options))
  write (u, "(2x,A)") &
    char (graph_options_get_gml_graphrange (g%graph_options))
  do i = 1, size (g%el)
    call graph_element_write_gml_reader (g%el(i), filename, unit)
    calc_cmd = drawing_options_get_calc_command &
      (graph_element_get_drawing_options_ptr (g%el(i)))
    if (calc_cmd /= "") write (u, "(2x,A)") char (calc_cmd)
  end do
  bg_cmd = graph_options_get_gml_bg_command (g%graph_options)
  if (bg_cmd /= "") write (u, "(2x,A)") char (bg_cmd)
  do i = 1, size (g%el)
    dro => graph_element_get_drawing_options_ptr (g%el(i))
    bg_cmd = drawing_options_get_gml_bg_command (dro)
    if (bg_cmd /= "") write (u, "(2x,A)") char (bg_cmd)
    draw_cmd = drawing_options_get_draw_command (dro)
    if (draw_cmd /= "") write (u, "(2x,A)") char (draw_cmd)
    err_cmd = drawing_options_get_err_command (dro)
    if (err_cmd /= "") write (u, "(2x,A)") char (err_cmd)
    symb_cmd = drawing_options_get_symb_command (dro)
    if (symb_cmd /= "") write (u, "(2x,A)") char (symb_cmd)
    fg_cmd = drawing_options_get_gml_fg_command (dro)
    if (fg_cmd /= "") write (u, "(2x,A)") char (fg_cmd)
  end do
  fg_cmd = graph_options_get_gml_fg_command (g%graph_options)
  if (fg_cmd /= "") write (u, "(2x,A)") char (fg_cmd)
  write (u, "(2x,A)") char (graph_options_get_gml_x_label (g%graph_options))
  write (u, "(2x,A)") char (graph_options_get_gml_y_label (g%graph_options))
  call graph_options_write_tex_footer (g%graph_options, unit)
end subroutine graph_write_gml_driver
```

Append header for generic data output in ifile format.

*<Analysis: procedures>+≡*

```
subroutine graph_get_header (graph, header, comment)
  type(graph_t), intent(in) :: graph
  type(ifile_t), intent(inout) :: header
```



```

type(string_t), intent(in), optional :: comment
type(string_t) :: c
if (present (comment)) then
    c = comment
else
    c = ""
end if
call ifile_append (header, c // "WHIZARD graph data")
call graph_options_get_header (graph%graph_options, header, comment)
call ifile_append (header, &
    c // "number of graph elements: " &
    // int2char (graph_get_n_elements (graph)))
end subroutine graph_get_header

```

### 5.1.9 Analysis objects

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings.

(This should rather be coded by type extension.)

```

<Analysis: parameters>≡
integer, parameter :: AN_UNDEFINED = 0
integer, parameter :: AN_OBSERVABLE = 1
integer, parameter :: AN_HISTOGRAM = 2
integer, parameter :: AN_PLOT = 3
integer, parameter :: AN_GRAPH = 4

<Analysis: public>+≡
public :: AN_UNDEFINED, AN_HISTOGRAM, AN_OBSERVABLE, AN_PLOT, AN_GRAPH

<Analysis: types>+≡
type :: analysis_object_t
private
type(string_t) :: id
integer :: type = AN_UNDEFINED
type(observable_t), pointer :: obs => null ()
type(histogram_t), pointer :: h => null ()
type(plot_t), pointer :: p => null ()
type(graph_t), pointer :: g => null ()
type(analysis_object_t), pointer :: next => null ()
end type analysis_object_t

```

#### Initializer/finalizer

Allocate with the correct type but do not fill initial values.

```

<Analysis: procedures>+≡
subroutine analysis_object_init (obj, id, type)
type(analysis_object_t), intent(out) :: obj
type(string_t), intent(in) :: id
integer, intent(in) :: type
obj%id = id

```

```

obj%type = type
select case (obj%type)
case (AN_OBSERVABLE); allocate (obj%obs)
case (AN_HISTOGRAM);  allocate (obj%h)
case (AN_PLOT);       allocate (obj%p)
case (AN_GRAPH);      allocate (obj%g)
end select
end subroutine analysis_object_init

```

*(Analysis: procedures)+≡*

```

subroutine analysis_object_final (obj)
type(analysis_object_t), intent(inout) :: obj
select case (obj%type)
case (AN_OBSERVABLE)
deallocate (obj%obs)
case (AN_HISTOGRAM)
deallocate (obj%h)
case (AN_PLOT)
call plot_final (obj%p)
deallocate (obj%p)
case (AN_GRAPH)
call graph_final (obj%g)
deallocate (obj%g)
end select
obj%type = AN_UNDEFINED
end subroutine analysis_object_final

```

Clear the analysis object, i.e., reset it to its initial state. Not applicable to graphs, which are always combinations of other existing objects.

*(Analysis: procedures)+≡*

```

subroutine analysis_object_clear (obj)
type(analysis_object_t), intent(inout) :: obj
select case (obj%type)
case (AN_OBSERVABLE)
call observable_clear (obj%obs)
case (AN_HISTOGRAM)
call histogram_clear (obj%h)
case (AN_PLOT)
call plot_clear (obj%p)
end select
end subroutine analysis_object_clear

```

## Fill with data

Record data. The effect depends on the type of analysis object.

*(Analysis: procedures)+≡*

```

subroutine analysis_object_record_data (obj, &
x, y, yerr, xerr, weight, excess, success)
type(analysis_object_t), intent(inout) :: obj
real(default), intent(in) :: x
real(default), intent(in), optional :: y, yerr, xerr, weight, excess

```

```

logical, intent(out), optional :: success
select case (obj%type)
case (AN_OBSERVABLE)
  if (present (weight)) then
    call observable_record_value_weighted (obj%obs, x, weight, success)
  else
    call observable_record_value_unweighted (obj%obs, x, success)
  end if
case (AN_HISTOGRAM)
  if (present (weight)) then
    call histogram_record_value_weighted (obj%h, x, weight, success)
  else
    call histogram_record_value_unweighted (obj%h, x, excess, success)
  end if
case (AN_PLOT)
  if (present (y)) then
    call plot_record_value (obj%p, x, y, yerr, xerr, success)
  else
    if (present (success)) success = .false.
  end if
case default
  if (present (success)) success = .false.
end select
end subroutine analysis_object_record_data

```

Explicitly set the pointer to the next object in the list.

```

<Analysis: procedures>+≡
subroutine analysis_object_set_next_ptr (obj, next)
  type(analysis_object_t), intent(inout) :: obj
  type(analysis_object_t), pointer :: next
  obj%next => next
end subroutine analysis_object_set_next_ptr

```

## Access contents

Return a pointer to the next object in the list.

```

<Analysis: procedures>+≡
function analysis_object_get_next_ptr (obj) result (next)
  type(analysis_object_t), pointer :: next
  type(analysis_object_t), intent(in) :: obj
  next => obj%next
end function analysis_object_get_next_ptr

```

Return data as appropriate for the object type.

```

<Analysis: procedures>+≡
function analysis_object_get_n_elements (obj) result (n)
  integer :: n
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_HISTOGRAM)
    n = 1

```

```

    case (AN_PLOT)
        n = 1
    case (AN_GRAPH)
        n = graph_get_n_elements (obj%g)
    case default
        n = 0
    end select
end function analysis_object_get_n_elements

function analysis_object_get_n_entries (obj, within_bounds) result (n)
    integer :: n
    type(analysis_object_t), intent(in) :: obj
    logical, intent(in), optional :: within_bounds
    logical :: wb
    select case (obj%type)
    case (AN_OBSERVABLE)
        n = observable_get_n_entries (obj%obs)
    case (AN_HISTOGRAM)
        wb = .false.; if (present (within_bounds)) wb = within_bounds
        if (wb) then
            n = histogram_get_n_entries_within_bounds (obj%h)
        else
            n = histogram_get_n_entries (obj%h)
        end if
    case (AN_PLOT)
        n = plot_get_n_entries (obj%p)
    case default
        n = 0
    end select
end function analysis_object_get_n_entries

function analysis_object_get_average (obj, within_bounds) result (avg)
    real(default) :: avg
    type(analysis_object_t), intent(in) :: obj
    logical, intent(in), optional :: within_bounds
    logical :: wb
    select case (obj%type)
    case (AN_OBSERVABLE)
        avg = observable_get_average (obj%obs)
    case (AN_HISTOGRAM)
        wb = .false.; if (present (within_bounds)) wb = within_bounds
        if (wb) then
            avg = histogram_get_average_within_bounds (obj%h)
        else
            avg = histogram_get_average (obj%h)
        end if
    case default
        avg = 0
    end select
end function analysis_object_get_average

function analysis_object_get_error (obj, within_bounds) result (err)
    real(default) :: err
    type(analysis_object_t), intent(in) :: obj

```

```

logical, intent(in), optional :: within_bounds
logical :: wb
select case (obj%type)
case (AN_OBSERVABLE)
    err = observable_get_error (obj%obs)
case (AN_HISTOGRAM)
    wb = .false.; if (present (within_bounds)) wb = within_bounds
    if (wb) then
        err = histogram_get_error_within_bounds (obj%h)
    else
        err = histogram_get_error (obj%h)
    end if
case default
    err = 0
end select
end function analysis_object_get_error

```

Return pointers to the actual contents:

*(Analysis: procedures)*+≡

```

function analysis_object_get_observable_ptr (obj) result (obs)
    type(observable_t), pointer :: obs
    type(analysis_object_t), intent(in) :: obj
    select case (obj%type)
    case (AN_OBSERVABLE); obs => obj%obs
    case default;         obs => null ()
    end select
end function analysis_object_get_observable_ptr

function analysis_object_get_histogram_ptr (obj) result (h)
    type(histogram_t), pointer :: h
    type(analysis_object_t), intent(in) :: obj
    select case (obj%type)
    case (AN_HISTOGRAM); h => obj%h
    case default;       h => null ()
    end select
end function analysis_object_get_histogram_ptr

function analysis_object_get_plot_ptr (obj) result (plot)
    type(plot_t), pointer :: plot
    type(analysis_object_t), intent(in) :: obj
    select case (obj%type)
    case (AN_PLOT); plot => obj%p
    case default;   plot => null ()
    end select
end function analysis_object_get_plot_ptr

function analysis_object_get_graph_ptr (obj) result (g)
    type(graph_t), pointer :: g
    type(analysis_object_t), intent(in) :: obj
    select case (obj%type)
    case (AN_GRAPH); g => obj%g
    case default;   g => null ()
    end select
end function analysis_object_get_graph_ptr

```

Return true if the object has a graphical representation:

```

<Analysis: procedures>+≡
function analysis_object_has_plot (obj) result (flag)
  logical :: flag
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
    case (AN_HISTOGRAM); flag = .true.
    case (AN_PLOT);      flag = .true.
    case (AN_GRAPH);     flag = .true.
    case default;        flag = .false.
  end select
end function analysis_object_has_plot

```

## Output

```

<Analysis: procedures>+≡
subroutine analysis_object_write (obj, unit)
  type(analysis_object_t), intent(in) :: obj
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(A)") repeat ("#", 79)
  select case (obj%type)
    case (AN_OBSERVABLE)
      write (u, "(A)", advance="no") "# Observable:"
    case (AN_HISTOGRAM)
      write (u, "(A)", advance="no") "# Histogram: "
    case (AN_PLOT)
      write (u, "(A)", advance="no") "# Plot: "
    case (AN_GRAPH)
      write (u, "(A)", advance="no") "# Graph: "
    case default
      write (u, "(A)") "# [undefined analysis object]"
  end select
  return
  write (u, "(1x,A)") char (obj%id)
  select case (obj%type)
    case (AN_OBSERVABLE); call observable_write (obj%obs, unit)
    case (AN_HISTOGRAM);  call histogram_write (obj%h, unit)
    case (AN_PLOT);       call plot_write (obj%p, unit)
    case (AN_GRAPH);      call graph_write (obj%g, unit)
  end select
end subroutine analysis_object_write

```

Write the object part of the L<sup>A</sup>T<sub>E</sub>X driver file.

```

<Analysis: procedures>+≡
subroutine analysis_object_write_driver (obj, filename, unit)
  type(analysis_object_t), intent(in) :: obj
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  select case (obj%type)

```

```

case (AN_OBSERVABLE)
  call observable_write_driver (obj%obs, unit)
case (AN_HISTOGRAM)
  call histogram_write_gml_driver (obj%h, filename, unit)
case (AN_PLOT)
  call plot_write_gml_driver (obj%p, filename, unit)
case (AN_GRAPH)
  call graph_write_gml_driver (obj%g, filename, unit)
end select
end subroutine analysis_object_write_driver

```

Return a data header for external formats, in ifile form.

```

<Analysis: procedures>+≡
subroutine analysis_object_get_header (obj, header, comment)
  type(analysis_object_t), intent(in) :: obj
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  select case (obj%type)
  case (AN_HISTOGRAM)
    call histogram_get_header (obj%h, header, comment)
  case (AN_PLOT)
    call plot_get_header (obj%p, header, comment)
  end select
end subroutine analysis_object_get_header

```

### 5.1.10 Analysis object iterator

Analysis objects are containers which have iterable data structures: histograms/bins and plots/points. If they are to be treated on a common basis, it is useful to have an iterator which hides the implementation details.

The iterator is used only for elementary analysis objects that contain plot data: histograms or plots. It is invalid for meta-objects (graphs) and non-graphical objects (observables).

```

<Analysis: public>+≡
  public :: analysis_iterator_t

<Analysis: types>+≡
  type :: analysis_iterator_t
  private
    integer :: type = AN_UNDEFINED
    type(analysis_object_t), pointer :: object => null ()
    integer :: index = 1
    type(point_t), pointer :: point => null ()
  end type

```

The initializer places the iterator at the beginning of the analysis object.

```

<Analysis: procedures>+≡
subroutine analysis_iterator_init (iterator, object)
  type(analysis_iterator_t), intent(out) :: iterator
  type(analysis_object_t), intent(in), target :: object
  iterator%object => object

```

```

    if (associated (iterator%object)) then
        iterator%type = iterator%object%type
        select case (iterator%type)
            case (AN_PLOT)
                iterator%point => iterator%object%p%first
            end select
        end if
    end if
end subroutine analysis_iterator_init

```

The iterator is valid as long as it points to an existing entry. An iterator for a data object without array data (observable) is always invalid.

```

<Analysis: public>+≡
    public :: analysis_iterator_is_valid

<Analysis: procedures>+≡
    function analysis_iterator_is_valid (iterator) result (valid)
        logical :: valid
        type(analysis_iterator_t), intent(in) :: iterator
        if (associated (iterator%object)) then
            select case (iterator%type)
                case (AN_HISTOGRAM)
                    valid = iterator%index <= histogram_get_n_bins (iterator%object%h)
                case (AN_PLOT)
                    valid = associated (iterator%point)
                case default
                    valid = .false.
            end select
        else
            valid = .false.
        end if
    end function analysis_iterator_is_valid

```

Advance the iterator.

```

<Analysis: public>+≡
    public :: analysis_iterator_advance

<Analysis: procedures>+≡
    subroutine analysis_iterator_advance (iterator)
        type(analysis_iterator_t), intent(inout) :: iterator
        if (associated (iterator%object)) then
            select case (iterator%type)
                case (AN_PLOT)
                    iterator%point => iterator%point%next
                end select
            iterator%index = iterator%index + 1
        end if
    end subroutine analysis_iterator_advance

```

Retrieve the object type:

```

<Analysis: public>+≡
    public :: analysis_iterator_get_type

```



```

<Analysis: procedures>+≡
function analysis_iterator_get_type (iterator) result (type)
integer :: type
type(analysis_iterator_t), intent(in) :: iterator
type = iterator%type
end function analysis_iterator_get_type

```

Use the iterator to retrieve data. We implement a common routine which takes the data descriptors as optional arguments. Data which do not occur in the selected type trigger to an error condition.

The iterator must point to a valid entry.

```

<Analysis: public>+≡
public :: analysis_iterator_get_data

<Analysis: procedures>+≡
subroutine analysis_iterator_get_data (iterator, &
x, y, yerr, xerr, width, excess, index, n_total)
type(analysis_iterator_t), intent(in) :: iterator
real(default), intent(out), optional :: x, y, yerr, xerr, width, excess
integer, intent(out), optional :: index, n_total
select case (iterator%type)
case (AN_HISTOGRAM)
if (present (x)) &
x = bin_get_midpoint (iterator%object%h%bin(iterator%index))
if (present (y)) &
y = bin_get_sum (iterator%object%h%bin(iterator%index))
if (present (yerr)) &
yerr = bin_get_error (iterator%object%h%bin(iterator%index))
if (present (xerr)) &
call invalid ("histogram", "xerr")
if (present (width)) &
width = bin_get_width (iterator%object%h%bin(iterator%index))
if (present (excess)) &
excess = bin_get_excess (iterator%object%h%bin(iterator%index))
if (present (index)) &
index = iterator%index
if (present (n_total)) &
n_total = histogram_get_n_bins (iterator%object%h)
case (AN_PLOT)
if (present (x)) &
x = point_get_x (iterator%point)
if (present (y)) &
y = point_get_y (iterator%point)
if (present (yerr)) &
yerr = point_get_yerr (iterator%point)
if (present (xerr)) &
xerr = point_get_xerr (iterator%point)
if (present (width)) &
call invalid ("plot", "width")
if (present (excess)) &
call invalid ("plot", "excess")
if (present (index)) &
index = iterator%index
if (present (n_total)) &

```

```

        n_total = plot_get_n_entries (iterator%object%p)
    case default
        call msg_bug ("analysis_iterator_get_data: called " &
            // "for unsupported analysis object type")
    end select
contains
    subroutine invalid (typestr, objstr)
        character(*), intent(in) :: typestr, objstr
        call msg_bug ("analysis_iterator_get_data: attempt to get '" &
            // objstr // "' for type '" // typestr // "'")
    end subroutine invalid
end subroutine analysis_iterator_get_data

```

### 5.1.11 Analysis store

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings and types.

```

<Analysis: variables>≡
    type(analysis_store_t), save :: analysis_store

<Analysis: types>+≡
    type :: analysis_store_t
    private
        type(analysis_object_t), pointer :: first => null ()
        type(analysis_object_t), pointer :: last => null ()
    end type analysis_store_t

```

Delete the analysis store

```

<Analysis: public>+≡
    public :: analysis_final

<Analysis: procedures>+≡
    subroutine analysis_final ()
        type(analysis_object_t), pointer :: current
        do while (associated (analysis_store%first))
            current => analysis_store%first
            analysis_store%first => current%next
            call analysis_object_final (current)
        end do
        analysis_store%last => null ()
    end subroutine analysis_final

```

Append a new analysis object

```

<Analysis: procedures>+≡
    subroutine analysis_store_append_object (id, type)
        type(string_t), intent(in) :: id
        integer, intent(in) :: type
        type(analysis_object_t), pointer :: obj
        allocate (obj)
        call analysis_object_init (obj, id, type)
    end subroutine analysis_store_append_object

```

```

    if (associated (analysis_store%last)) then
        analysis_store%last%next => obj
    else
        analysis_store%first => obj
    end if
    analysis_store%last => obj
end subroutine analysis_store_append_object

```

Return a pointer to the analysis object with given ID.

*<Analysis: procedures>+≡*

```

function analysis_store_get_object_ptr (id) result (obj)
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    obj => analysis_store%first
    do while (associated (obj))
        if (obj%id == id) return
        obj => obj%next
    end do
end function analysis_store_get_object_ptr

```

Initialize an analysis object: either reset it if present, or append a new entry.

*<Analysis: procedures>+≡*

```

subroutine analysis_store_init_object (id, type, obj)
    type(string_t), intent(in) :: id
    integer, intent(in) :: type
    type(analysis_object_t), pointer :: obj, next
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
        next => analysis_object_get_next_ptr (obj)
        call analysis_object_final (obj)
        call analysis_object_init (obj, id, type)
        call analysis_object_set_next_ptr (obj, next)
    else
        call analysis_store_append_object (id, type)
        obj => analysis_store%last
    end if
end subroutine analysis_store_init_object

```

Get the type of a analysis object

*<Analysis: public>+≡*

```

public :: analysis_store_get_object_type

```

*<Analysis: procedures>+≡*

```

function analysis_store_get_object_type (id) result (type)
    type(string_t), intent(in) :: id
    integer :: type
    type(analysis_object_t), pointer :: object
    object => analysis_store_get_object_ptr (id)
    if (associated (object)) then
        type = object%type
    else
        type = AN_UNDEFINED
    end if
end function

```

```
end function analysis_store_get_object_type
```

Return the number of objects in the store.

```
<Analysis: procedures>+≡
function analysis_store_get_n_objects () result (n)
  integer :: n
  type(analysis_object_t), pointer :: current
  n = 0
  current => analysis_store%first
  do while (associated (current))
    n = n + 1
    current => current%next
  end do
end function analysis_store_get_n_objects
```

Allocate an array and fill it with all existing IDs.

```
<Analysis: public>+≡
public :: analysis_store_get_ids

<Analysis: procedures>+≡
subroutine analysis_store_get_ids (id)
  type(string_t), dimension(:), allocatable, intent(out) :: id
  type(analysis_object_t), pointer :: current
  integer :: i
  allocate (id (analysis_store_get_n_objects()))
  i = 0
  current => analysis_store%first
  do while (associated (current))
    i = i + 1
    id(i) = current%id
    current => current%next
  end do
end subroutine analysis_store_get_ids
```

### 5.1.12 L<sup>A</sup>T<sub>E</sub>X driver file

Write a driver file for all objects in the store.

```
<Analysis: procedures>+≡
subroutine analysis_store_write_driver_all (filename_data, unit)
  type(string_t), intent(in) :: filename_data
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  call analysis_store_write_driver_header (unit)
  obj => analysis_store%first
  do while (associated (obj))
    call analysis_object_write_driver (obj, filename_data, unit)
    obj => obj%next
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_all
```

Write a driver file for an array of objects.

*<Analysis: procedures>+≡*

```
subroutine analysis_store_write_driver_obj (filename_data, id, unit)
  type(string_t), intent(in) :: filename_data
  type(string_t), dimension(:), intent(in) :: id
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  integer :: i
  call analysis_store_write_driver_header (unit)
  do i = 1, size (id)
    obj => analysis_store_get_object_ptr (id(i))
    if (associated (obj)) &
      call analysis_object_write_driver (obj, filename_data, unit)
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_obj
```

The beginning of the driver file.

*<Analysis: procedures>+≡*

```
subroutine analysis_store_write_driver_header (unit)
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write(u, '(A)') "\documentclass[12pt]{article}"
  write(u, *)
  write(u, '(A)') "\usepackage{gamelan}"
  write(u, '(A)') "\usepackage{amsmath}"
  write(u, *)
  write(u, '(A)') "\begin{document}"
  write(u, '(A)') "\begin{gmlfile}"
  write(u, *)
  write(u, '(A)') "\begin{gmlcode}"
  write(u, '(A)') "  color col.default, col.excess;"
  write(u, '(A)') "  col.default = 0.9white;"
  write(u, '(A)') "  col.excess = red;"
  write(u, '(A)') "  boolean show_excess;"
!   if (mcs(1)%plot_excess .and. mcs(1)%unweighted) then
!     write(u, '(A)') "  show_excess = true;"
!   else
  write(u, '(A)') "  show_excess = false;"
!   end if
  write(u, '(A)') "\end{gmlcode}"
  write(u, *)
end subroutine analysis_store_write_driver_header
```

The end of the driver file.

*<Analysis: procedures>+≡*

```
subroutine analysis_store_write_driver_footer (unit)
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write(u, *)
  write(u, '(A)') "\end{gmlfile}"
```

```

        write(u, '(A)') "\end{document}"
    end subroutine analysis_store_write_driver_footer

```

### 5.1.13 API

#### Creating new objects

The specific versions below:

```

<Analysis: public>+≡
    public :: analysis_init_observable

<Analysis: procedures>+≡
    subroutine analysis_init_observable (id, obs_label, obs_unit, graph_options)
        type(string_t), intent(in) :: id
        type(string_t), intent(in), optional :: obs_label, obs_unit
        type(graph_options_t), intent(in), optional :: graph_options
        type(analysis_object_t), pointer :: obj
        type(observable_t), pointer :: obs
        call analysis_store_init_object (id, AN_OBSERVABLE, obj)
        obs => analysis_object_get_observable_ptr (obj)
        call observable_init (obs, obs_label, obs_unit, graph_options)
    end subroutine analysis_init_observable

<Analysis: public>+≡
    public :: analysis_init_histogram

<Analysis: interfaces>+≡
    interface analysis_init_histogram
        module procedure analysis_init_histogram_n_bins
        module procedure analysis_init_histogram_bin_width
    end interface

<Analysis: procedures>+≡
    subroutine analysis_init_histogram_n_bins &
        (id, lower_bound, upper_bound, n_bins, normalize_bins, &
         obs_label, obs_unit, graph_options, drawing_options)
        type(string_t), intent(in) :: id
        real(default), intent(in) :: lower_bound, upper_bound
        integer, intent(in) :: n_bins
        logical, intent(in) :: normalize_bins
        type(string_t), intent(in), optional :: obs_label, obs_unit
        type(graph_options_t), intent(in), optional :: graph_options
        type(drawing_options_t), intent(in), optional :: drawing_options
        type(analysis_object_t), pointer :: obj
        type(histogram_t), pointer :: h
        call analysis_store_init_object (id, AN_HISTOGRAM, obj)
        h => analysis_object_get_histogram_ptr (obj)
        call histogram_init (h, id, &
            lower_bound, upper_bound, n_bins, normalize_bins, &
            obs_label, obs_unit, graph_options, drawing_options)
    end subroutine analysis_init_histogram_n_bins

    subroutine analysis_init_histogram_bin_width &

```

```

        (id, lower_bound, upper_bound, bin_width, normalize_bins, &
         obs_label, obs_unit, graph_options, drawing_options)
type(string_t), intent(in) :: id
real(default), intent(in) :: lower_bound, upper_bound, bin_width
logical, intent(in) :: normalize_bins
type(string_t), intent(in), optional :: obs_label, obs_unit
type(graph_options_t), intent(in), optional :: graph_options
type(drawing_options_t), intent(in), optional :: drawing_options
type(analysis_object_t), pointer :: obj
type(histogram_t), pointer :: h
call analysis_store_init_object (id, AN_HISTOGRAM, obj)
h => analysis_object_get_histogram_ptr (obj)
call histogram_init (h, id, &
                    lower_bound, upper_bound, bin_width, normalize_bins, &
                    obs_label, obs_unit, graph_options, drawing_options)
end subroutine analysis_init_histogram_bin_width

```

*<Analysis: public>+≡*

```
public :: analysis_init_plot
```

*<Analysis: procedures>+≡*

```

subroutine analysis_init_plot (id, graph_options, drawing_options)
type(string_t), intent(in) :: id
type(graph_options_t), intent(in), optional :: graph_options
type(drawing_options_t), intent(in), optional :: drawing_options
type(analysis_object_t), pointer :: obj
type(plot_t), pointer :: plot
call analysis_store_init_object (id, AN_PLOT, obj)
plot => analysis_object_get_plot_ptr (obj)
call plot_init (plot, id, graph_options, drawing_options)
end subroutine analysis_init_plot

```

*<Analysis: public>+≡*

```
public :: analysis_init_graph
```

*<Analysis: procedures>+≡*

```

subroutine analysis_init_graph (id, n_elements, graph_options)
type(string_t), intent(in) :: id
integer, intent(in) :: n_elements
type(graph_options_t), intent(in), optional :: graph_options
type(analysis_object_t), pointer :: obj
type(graph_t), pointer :: graph
call analysis_store_init_object (id, AN_GRAPH, obj)
graph => analysis_object_get_graph_ptr (obj)
call graph_init (graph, id, n_elements, graph_options)
end subroutine analysis_init_graph

```

## Recording data

This procedure resets an object or the whole store to its initial state.

*<Analysis: public>+≡*

```
public :: analysis_clear
```

```

<Analysis: interfaces>+≡
  interface analysis_clear
    module procedure analysis_store_clear_obj
    module procedure analysis_store_clear_all
  end interface

```

```

<Analysis: procedures>+≡
  subroutine analysis_store_clear_obj (id)
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
      call analysis_object_clear (obj)
    end if
  end subroutine analysis_store_clear_obj

  subroutine analysis_store_clear_all ()
    type(analysis_object_t), pointer :: obj
    obj => analysis_store%first
    do while (associated (obj))
      call analysis_object_clear (obj)
      obj => obj%next
    end do
  end subroutine analysis_store_clear_all

```

There is one generic recording function whose behavior depends on the type of analysis object.

```

<Analysis: public>+≡
  public :: analysis_record_data

<Analysis: procedures>+≡
  subroutine analysis_record_data (id, x, y, yerr, xerr, &
    weight, excess, success, exist)
    type(string_t), intent(in) :: id
    real(default), intent(in) :: x
    real(default), intent(in), optional :: y, yerr, xerr, weight, excess
    logical, intent(out), optional :: success, exist
    type(analysis_object_t), pointer :: obj
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
      call analysis_object_record_data (obj, x, y, yerr, xerr, &
        weight, excess, success)
      if (present (exist)) exist = .true.
    else
      if (present (success)) success = .false.
      if (present (exist)) exist = .false.
    end if
  end subroutine analysis_record_data

```



## Build a graph

This routine sets up the array of graph elements by copying the graph elements given as input. The object must exist and already be initialized as a graph.

*<Analysis: public>+≡*

```
public :: analysis_fill_graph
```

*<Analysis: procedures>+≡*

```
subroutine analysis_fill_graph (id, i, id_in, drawing_options)
  type(string_t), intent(in) :: id
  integer, intent(in) :: i
  type(string_t), intent(in) :: id_in
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(analysis_object_t), pointer :: obj
  type(graph_t), pointer :: g
  type(histogram_t), pointer :: h
  type(plot_t), pointer :: p
  obj => analysis_store_get_object_ptr (id)
  g => analysis_object_get_graph_ptr (obj)
  obj => analysis_store_get_object_ptr (id_in)
  if (associated (obj)) then
    select case (obj%type)
    case (AN_HISTOGRAM)
      h => analysis_object_get_histogram_ptr (obj)
      call graph_insert_histogram (g, i, h, drawing_options)
    case (AN_PLOT)
      p => analysis_object_get_plot_ptr (obj)
      call graph_insert_plot (g, i, p, drawing_options)
    case default
      call msg_error ("Graph '" // char (id) // "': Element '" &
        // char (id_in) // "' is neither histogram nor plot.")
    end select
  else
    call msg_error ("Graph '" // char (id) // "': Element '" &
      // char (id_in) // "' is undefined.")
  end if
end subroutine analysis_fill_graph
```

## Retrieve generic results

The following functions should work for all kinds of analysis object:

*<Analysis: public>+≡*

```
public :: analysis_get_n_elements
public :: analysis_get_n_entries
public :: analysis_get_average
public :: analysis_get_error
```

*<Analysis: procedures>+≡*

```
function analysis_get_n_elements (id) result (n)
  integer :: n
  type(string_t), intent(in) :: id
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) then
```

```

        n = analysis_object_get_n_elements (obj)
    else
        n = 0
    end if
end function analysis_get_n_elements

function analysis_get_n_entries (id, within_bounds) result (n)
    integer :: n
    type(string_t), intent(in) :: id
    logical, intent(in), optional :: within_bounds
    type(analysis_object_t), pointer :: obj
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
        n = analysis_object_get_n_entries (obj, within_bounds)
    else
        n = 0
    end if
end function analysis_get_n_entries

function analysis_get_average (id, within_bounds) result (avg)
    real(default) :: avg
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
        avg = analysis_object_get_average (obj, within_bounds)
    else
        avg = 0
    end if
end function analysis_get_average

function analysis_get_error (id, within_bounds) result (err)
    real(default) :: err
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
        err = analysis_object_get_error (obj, within_bounds)
    else
        err = 0
    end if
end function analysis_get_error

```

Return true if any analysis object is graphical

*<Analysis: public>+≡*

```
public :: analysis_has_plots
```

*<Analysis: interfaces>+≡*

```
interface analysis_has_plots
    module procedure analysis_has_plots_any
    module procedure analysis_has_plots_obj
end interface

```

*<Analysis: procedures>+≡*

```

function analysis_has_plots_any () result (flag)
  logical :: flag
  type(analysis_object_t), pointer :: obj
  flag = .false.
  obj => analysis_store%first
  do while (associated (obj))
    flag = analysis_object_has_plot (obj)
    if (flag) return
  end do
end function analysis_has_plots_any

function analysis_has_plots_obj (id) result (flag)
  logical :: flag
  type(string_t), dimension(:), intent(in) :: id
  type(analysis_object_t), pointer :: obj
  integer :: i
  flag = .false.
  do i = 1, size (id)
    obj => analysis_store_get_object_ptr (id(i))
    if (associated (obj)) then
      flag = analysis_object_has_plot (obj)
      if (flag) return
    end if
  end do
end function analysis_has_plots_obj

```

## Iterators

Initialize an iterator for the given object. If the object does not exist or has wrong type, the iterator will be invalid.

*<Analysis: public>+≡*

```

public :: analysis_init_iterator

```

*<Analysis: procedures>+≡*

```

subroutine analysis_init_iterator (id, iterator)
  type(string_t), intent(in) :: id
  type(analysis_iterator_t), intent(out) :: iterator
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) call analysis_iterator_init (iterator, obj)
end subroutine analysis_init_iterator

```

## Output

*<Analysis: public>+≡*

```

public :: analysis_write

```

```

<Analysis: interfaces>+≡
    interface analysis_write
        module procedure analysis_write_object
        module procedure analysis_write_all
    end interface

<Analysis: procedures>+≡
    subroutine analysis_write_object (id, unit)
        type(string_t), intent(in) :: id
        integer, intent(in), optional :: unit
        type(analysis_object_t), pointer :: obj
        obj => analysis_store_get_object_ptr (id)
        if (associated (obj)) then
            call analysis_object_write (obj, unit)
        else
            call msg_error ("Analysis object '" // char (id) // "' not found")
        end if
    end subroutine analysis_write_object

    subroutine analysis_write_all (unit)
        integer, intent(in), optional :: unit
        type(analysis_object_t), pointer :: obj
        integer :: u
        u = output_unit (unit); if (u < 0) return
        obj => analysis_store%first
        do while (associated (obj))
            call analysis_object_write (obj, unit)
            obj => obj%next
        end do
    end subroutine analysis_write_all

<Analysis: public>+≡
    public :: analysis_write_driver

<Analysis: procedures>+≡
    subroutine analysis_write_driver (filename_data, id, unit)
        type(string_t), intent(in) :: filename_data
        type(string_t), dimension(:), intent(in), optional :: id
        integer, intent(in), optional :: unit
        if (present (id)) then
            call analysis_store_write_driver_obj (filename_data, id, unit)
        else
            call analysis_store_write_driver_all (filename_data, unit)
        end if
    end subroutine analysis_write_driver

<Analysis: public>+≡
    public :: analysis_compile_tex

<Analysis: procedures>+≡
    subroutine analysis_compile_tex (file, has_gmlcode, os_data)
        type(string_t), intent(in) :: file
        logical, intent(in) :: has_gmlcode
        type(os_data_t), intent(in) :: os_data

```

```

type(string_t) :: setenv
integer :: status
if (os_data%event_analysis_ps) then
  BLOCK: do
    if (os_data%whizard_texpath /= "") then
      setenv = "TEXINPUTS=" // os_data%whizard_texpath // ":$TEXINPUTS "
    else
      setenv = ""
    end if
    call os_system_call (setenv // os_data%latex // " " // file, status)
    if (status /= 0) exit BLOCK
    if (has_gmlcode) then
      call os_system_call (os_data%gml // " " // file, status)
      if (status /= 0) exit BLOCK
      call os_system_call (setenv // os_data%latex // " " // file, &
        status)
      if (status /= 0) exit BLOCK
    end if
    call os_system_call (os_data%dvips // " -o " // file // ".ps " &
      // file, status)
    if (status /= 0) exit BLOCK
    if (os_data%event_analysis_pdf) then
      call os_system_call (os_data%ps2pdf // " " // file // ".ps", &
        status)
      if (status /= 0) exit BLOCK
    end if
    exit BLOCK
  end do BLOCK
  if (status /= 0) then
    call msg_error ("Unable to compile analysis output file")
  end if
else
  call msg_warning ("Skipping results display because " &
    // "latex/mpost/dvips is not available")
end if
end subroutine analysis_compile_tex

```

Write header for generic data output to an ifile.

*<Analysis: public>+≡*

```
public :: analysis_get_header
```

*<Analysis: procedures>+≡*

```

subroutine analysis_get_header (id, header, comment)
  type(string_t), intent(in) :: id
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  type(analysis_object_t), pointer :: object
  object => analysis_store_get_object_ptr (id)
  if (associated (object)) then
    call analysis_object_get_header (object, header, comment)
  end if
end subroutine analysis_get_header

```

### 5.1.14 Test

```
<Analysis: public>+≡
    public :: analysis_test

<Analysis: procedures>+≡
    subroutine analysis_test ()
        call analysis_test1 ()
        call analysis_final ()
    end subroutine analysis_test

<Analysis: procedures>+≡
    subroutine analysis_test1 ()
        type(string_t) :: id1, id2, id3, id4
        integer :: i
        id1 = "foo"
        id2 = "bar"
        id3 = "hist"
        id4 = "plot"
        call analysis_init_observable (id1)
        call analysis_init_observable (id2)
        call analysis_init_histogram_bin_width &
            (id3, 0.5_default, 5.5_default, 1._default, normalize_bins=.false.)
        call analysis_init_plot (id4)
        do i = 1, 3
            print *, "data = ", real(i,default)
            call analysis_record_data (id1, real(i,default))
            call analysis_record_data (id2, real(i,default), &
                weight=real(i,default))
            call analysis_record_data (id3, real(i,default))
            call analysis_record_data (id4, real(i,default), real(i,default)**2)
        end do
1   format (A,10(1x,I5))
2   format (A,10(1x,F5.3))
        print 1, "n_entries = ", &
            analysis_get_n_entries (id1), &
            analysis_get_n_entries (id2), &
            analysis_get_n_entries (id3), &
            analysis_get_n_entries (id3, within_bounds = .true.), &
            analysis_get_n_entries (id4), &
            analysis_get_n_entries (id4, within_bounds = .true.)
        print 2, "average   = ", &
            analysis_get_average (id1), &
            analysis_get_average (id2), &
            analysis_get_average (id3), &
            analysis_get_average (id3, within_bounds = .true.)
        print 2, "error     = ", &
            analysis_get_error (id1), &
            analysis_get_error (id2), &
            analysis_get_error (id3), &
            analysis_get_error (id3, within_bounds = .true.)
        print *, "clear #2"
        call analysis_clear (id2)
        do i = 4, 6
            print *, "data = ", real(i,default)
```

```

      call analysis_record_data (id1, real(i,default))
      call analysis_record_data (id2, real(i,default), &
                                weight=real(i,default))
      call analysis_record_data (id3, real(i,default))
      call analysis_record_data (id4, real(i,default), real(i,default)**2)
end do
print 1, "n_entries = ", &
      analysis_get_n_entries (id1), &
      analysis_get_n_entries (id2), &
      analysis_get_n_entries (id3), &
      analysis_get_n_entries (id3, within_bounds = .true.), &
      analysis_get_n_entries (id4), &
      analysis_get_n_entries (id4, within_bounds = .true.)
print 2, "average   = ", &
      analysis_get_average (id1), &
      analysis_get_average (id2), &
      analysis_get_average (id3), &
      analysis_get_average (id3, within_bounds = .true.)
print 2, "error     = ", &
      analysis_get_error (id1), &
      analysis_get_error (id2), &
      analysis_get_error (id3), &
      analysis_get_error (id3, within_bounds = .true.)
print *
call analysis_write ()
call analysis_clear ()
end subroutine analysis_test1

```

## 5.2 PDG arrays

For defining aliases, we introduce a special type which holds a set of (integer) PDG codes.

```
<pdg_arrays.f90>≡  
<File header>  
  
module pdg_arrays  
  
<Use file utils>  
  
<Standard module head>  
  
<PDG arrays: public>  
  
<PDG arrays: parameters>  
  
<PDG arrays: types>  
  
<PDG arrays: interfaces>  
  
contains  
  
<PDG arrays: procedures>  
  
end module pdg_arrays
```

### 5.2.1 Type definition

Using an allocatable array eliminates the need for initializer and/or finalizer.

```
<PDG arrays: public>≡  
public :: pdg_array_t  
  
<PDG arrays: types>≡  
type :: pdg_array_t  
private  
integer, dimension(:), allocatable :: pdg  
end type pdg_array_t
```

Output

```
<PDG arrays: public>+≡  
public :: pdg_array_write  
  
<PDG arrays: procedures>≡  
subroutine pdg_array_write (aval, unit)  
type(pdg_array_t), intent(in) :: aval  
integer, intent(in), optional :: unit  
integer :: u, i  
u = output_unit (unit); if (u < 0) return  
write (u, "(A)", advance="no") "PDG("  
if (allocated (aval%pdg)) then  
do i = 1, size (aval%pdg)  
if (i > 1) write (u, "(A)", advance="no") " , "
```



```

        write (u, "(I0)", advance="no")  aval%pdg(i)
    end do
end if
write (u, "(A)", advance="no")  ")"
end subroutine pdg_array_write

```

### 5.2.2 Parameters

We need an UNDEFINED value:

```

(PDG arrays: parameters)≡
    integer, parameter, public :: UNDEFINED = 0

```

### 5.2.3 Basic operations

Assignment. We define assignment from and to an integer array. Note that the integer array, if it is the l.h.s., must be declared allocatable by the caller.

```

(PDG arrays: public)+≡
    public :: assignment(=)

(PDG arrays: interfaces)≡
    interface assignment(=)
        module procedure pdg_array_from_int_array
        module procedure pdg_array_from_int
        module procedure int_array_from_pdg_array
    end interface

(PDG arrays: procedures)+≡
    subroutine pdg_array_from_int_array (aval, iarray)
        type(pdg_array_t), intent(out) :: aval
        integer, dimension(:), intent(in) :: iarray
        allocate (aval%pdg (size (iarray)))
        aval%pdg = iarray
    end subroutine pdg_array_from_int_array

    elemental subroutine pdg_array_from_int (aval, int)
        type(pdg_array_t), intent(out) :: aval
        integer, intent(in) :: int
        allocate (aval%pdg (1))
        aval%pdg = int
    end subroutine pdg_array_from_int

    subroutine int_array_from_pdg_array (iarray, aval)
        integer, dimension(:), allocatable, intent(out) :: iarray
        type(pdg_array_t), intent(in) :: aval
        if (allocated (aval%pdg)) then
            allocate (iarray (size (aval%pdg)))
            iarray = aval%pdg
        else
            allocate (iarray (0))
        end if
    end subroutine int_array_from_pdg_array

```

The only nontrivial operation: concatenate two PDG arrays

```

(PDG arrays: public)+≡
    public :: operator(//)

(PDG arrays: interfaces)+≡
    interface operator(//)
        module procedure concat_pdg_arrays
    end interface

(PDG arrays: procedures)+≡
    function concat_pdg_arrays (aval1, aval2) result (aval)
        type(pdg_array_t) :: aval
        type(pdg_array_t), intent(in) :: aval1, aval2
        integer :: n1, n2
        if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
            n1 = size (aval1%pdg)
            n2 = size (aval2%pdg)
            allocate (aval%pdg (n1 + n2))
            aval%pdg(:n1) = aval1%pdg
            aval%pdg(n1+1:) = aval2%pdg
        else if (allocated (aval1%pdg)) then
            aval = aval1
        else if (allocated (aval2%pdg)) then
            aval = aval2
        end if
    end function concat_pdg_arrays

```

#### 5.2.4 Matching

A PDG array matches a given PDG code if the code is present within the array. If either one is zero (UNDEFINED), the match also succeeds.

```

(PDG arrays: public)+≡
    public :: operator(.match.)

(PDG arrays: interfaces)+≡
    interface operator(.match.)
        module procedure pdg_array_match_integer
    end interface

(PDG arrays: procedures)+≡
    elemental function pdg_array_match_integer (aval, pdg) result (flag)
        logical :: flag
        type(pdg_array_t), intent(in) :: aval
        integer, intent(in) :: pdg
        if (allocated (aval%pdg)) then
            flag = pdg == UNDEFINED &
                .or. any (aval%pdg == UNDEFINED) &
                .or. any (aval%pdg == pdg)
        else
            flag = .false.
        end if
    end function pdg_array_match_integer

```

Equivalence. Two PDG arrays are equivalent if either both contain UNDEFINED or each element of array 1 is present in array 2, and vice versa.

```

(PDG arrays: public)+≡
    public :: operator(.eqv.)
    public :: operator(.neqv.)

(PDG arrays: interfaces)+≡
    interface operator(.eqv.)
        module procedure pdg_array_equivalent
    end interface
    interface operator(.neqv.)
        module procedure pdg_array_inequivalent
    end interface

(PDG arrays: procedures)+≡
    function pdg_array_equivalent (aval1, aval2) result (eq)
        logical :: eq
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical, dimension(:), allocatable :: match1, match2
        integer :: i
        if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
            eq = any (aval1%pdg == UNDEFINED) &
                .and. any (aval2%pdg == UNDEFINED)
            if (.not. eq) then
                allocate (match1 (size (aval1%pdg)))
                allocate (match2 (size (aval2%pdg)))
                match1 = .false.
                match2 = .false.
                do i = 1, size (aval1%pdg)
                    match2 = match2 .or. aval1%pdg(i) == aval2%pdg
                end do
                do i = 1, size (aval2%pdg)
                    match1 = match1 .or. aval2%pdg(i) == aval1%pdg
                end do
                eq = all (match1) .and. all (match2)
            end if
        else
            eq = .false.
        end if
    end function pdg_array_equivalent

    function pdg_array_inequivalent (aval1, aval2) result (neq)
        logical :: neq
        type(pdg_array_t), intent(in) :: aval1, aval2
        neq = .not. pdg_array_equivalent (aval1, aval2)
    end function pdg_array_inequivalent

```

## 5.3 subevents

The purpose of subevents is to store the relevant part of the physical event (either partonic or hadronic), and to hold particle selections and combinations which are constructed in cut or analysis expressions.

```
<subevents.f90>≡  
  <File header>  
  
  module subevents  
  
    use iso_c_binding !NODEP!  
    <Use kinds>  
    <Use file utils>  
    use c_particles !NODEP!  
    use lorentz !NODEP!  
    use sorting  
    use pdg_arrays  
  
    <Standard module head>  
  
    <Subevents: public>  
  
    <Subevents: parameters>  
  
    <Subevents: types>  
  
    <Subevents: interfaces>  
  
    contains  
  
    <Subevents: procedures>  
  
  end module subevents
```

### 5.3.1 Particles

For the purpose of this module, a particle has a type which can indicate a beam, incoming, outgoing, or composite particle, flavor and helicity codes (integer, undefined for composite), four-momentum and invariant mass squared. (Other particles types are used in extended event types, but also defined here.) Furthermore, each particle has an allocatable array of ancestors – particle indices which indicate the building blocks of a composite particle. For an incoming/outgoing particle, the array contains only the index of the particle itself.

For incoming particles, the momentum is inverted before storing it in the particle object.

```
<Subevents: parameters>≡  
  integer, parameter, public :: PRT_UNDEFINED = 0  
  integer, parameter, public :: PRT_BEAM = -9  
  integer, parameter, public :: PRT_INCOMING = 1  
  integer, parameter, public :: PRT_OUTGOING = 2  
  integer, parameter, public :: PRT_COMPOSITE = 3  
  integer, parameter, public :: PRT_VIRTUAL = 4
```

```

integer, parameter, public :: PRT_RESONANT = 5
integer, parameter, public :: PRT_BEAM_REMNANT = 9

```

## The type

We initialize only the type here and mark as unpolarized. The initializers below do the rest.

```

<Subevents: public>≡
    public :: prt_t

<Subevents: types>≡
    type :: prt_t
        private
        integer :: type = PRT_UNDEFINED
        integer :: pdg
        logical :: polarized = .false.
        integer :: h
        type(vector4_t) :: p
        real(default) :: p2
        integer, dimension(:), allocatable :: src
    end type prt_t

```

Initializers. Polarization is set separately. Finalizers are not needed.

```

<Subevents: procedures>≡
    subroutine prt_init_beam (prt, pdg, p, p2, src)
        type(prt_t), intent(out) :: prt
        integer, intent(in) :: pdg
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: p2
        integer, dimension(:), intent(in) :: src
        prt%type = PRT_BEAM
        call prt_set (prt, pdg, - p, p2, src)
    end subroutine prt_init_beam

    subroutine prt_init_incoming (prt, pdg, p, p2, src)
        type(prt_t), intent(out) :: prt
        integer, intent(in) :: pdg
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: p2
        integer, dimension(:), intent(in) :: src
        prt%type = PRT_INCOMING
        call prt_set (prt, pdg, - p, p2, src)
    end subroutine prt_init_incoming

    subroutine prt_init_outgoing (prt, pdg, p, p2, src)
        type(prt_t), intent(out) :: prt
        integer, intent(in) :: pdg
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: p2
        integer, dimension(:), intent(in) :: src
        prt%type = PRT_OUTGOING
        call prt_set (prt, pdg, p, p2, src)
    end subroutine prt_init_outgoing

```

```

end subroutine prt_init_outgoing

subroutine prt_init_composite (prt, p, src)
  type(prt_t), intent(out) :: prt
  type(vector4_t), intent(in) :: p
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_COMPOSITE
  call prt_set (prt, 0, p, p**2, src)
end subroutine prt_init_composite

```

This version is for temporary particle objects, so the `src` array is not set.

```

<Subevents: public>+≡
  public :: prt_init_combine

<Subevents: procedures>+≡
  subroutine prt_init_combine (prt, prt1, prt2)
    type(prt_t), intent(out) :: prt
    type(prt_t), intent(in) :: prt1, prt2
    type(vector4_t) :: p
    integer, dimension(0) :: src
    prt%type = PRT_COMPOSITE
    p = prt1%p + prt2%p
    call prt_set (prt, 0, p, p**2, src)
  end subroutine prt_init_combine

```

## Accessing contents

```

<Subevents: public>+≡
  public :: prt_get_pdg

<Subevents: procedures>+≡
  elemental function prt_get_pdg (prt) result (pdg)
    integer :: pdg
    type(prt_t), intent(in) :: prt
    pdg = prt%pdg
  end function prt_get_pdg

<Subevents: public>+≡
  public :: prt_get_momentum

<Subevents: procedures>+≡
  elemental function prt_get_momentum (prt) result (p)
    type(vector4_t) :: p
    type(prt_t), intent(in) :: prt
    p = prt%p
  end function prt_get_momentum

<Subevents: public>+≡
  public :: prt_get_msq

<Subevents: procedures>+≡
  elemental function prt_get_msq (prt) result (msq)
    real(default) :: msq
    type(prt_t), intent(in) :: prt

```

```

        msq = prt%p2
    end function prt_get_msq

<Subevents: public>+≡
    public :: prt_is_polarized

<Subevents: procedures>+≡
    elemental function prt_is_polarized (prt) result (flag)
        logical :: flag
        type(prt_t), intent(in) :: prt
        flag = prt%polarized
    end function prt_is_polarized

<Subevents: public>+≡
    public :: prt_get_helicity

<Subevents: procedures>+≡
    elemental function prt_get_helicity (prt) result (h)
        integer :: h
        type(prt_t), intent(in) :: prt
        h = prt%h
    end function prt_get_helicity

```

## Setting data

Set the PDG, momentum and momentum squared, and ancestors. If allocation-assignment is available, this can be simplified.

```

<Subevents: procedures>+≡
    subroutine prt_set (prt, pdg, p, p2, src)
        type(prt_t), intent(inout) :: prt
        integer, intent(in) :: pdg
        type(vector4_t), intent(in) :: p
        real(default), intent(in) :: p2
        integer, dimension(:), intent(in) :: src
        prt%pdg = pdg
        prt%p = p
        prt%p2 = p2
        if (allocated (prt%src)) then
            if (size (src) /= size (prt%src)) then
                deallocate (prt%src)
                allocate (prt%src (size (src)))
            end if
        else
            allocate (prt%src (size (src)))
        end if
        prt%src = src
    end subroutine prt_set

```

Set the particle PDG code separately.

```

<Subevents: procedures>+≡
    elemental subroutine prt_set_pdg (prt, pdg)
        type(prt_t), intent(inout) :: prt

```

```

        integer, intent(in) :: pdg
        prt%pdg = pdg
    end subroutine prt_set_pdg

```

Set the momentum separately.

```

<Subevents: procedures>+≡
    elemental subroutine prt_set_p (prt, p)
        type(prt_t), intent(inout) :: prt
        type(vector4_t), intent(in) :: p
        prt%p = p
    end subroutine prt_set_p

```

Set the squared invariant mass separately.

```

<Subevents: procedures>+≡
    elemental subroutine prt_set_p2 (prt, p2)
        type(prt_t), intent(inout) :: prt
        real(default), intent(in) :: p2
        prt%p2 = p2
    end subroutine prt_set_p2

```

Set helicity (optional).

```

<Subevents: procedures>+≡
    subroutine prt_polarize (prt, h)
        type(prt_t), intent(inout) :: prt
        integer, intent(in) :: h
        prt%polarized = .true.
        prt%h = h
    end subroutine prt_polarize

```

## Conversion

Transform a `prt_t` object into a `c_prt_t` object.

```

<Subevents: public>+≡
    public :: c_prt

<Subevents: interfaces>≡
    interface c_prt
        module procedure c_prt_from_prt
    end interface

<Subevents: procedures>+≡
    elemental function c_prt_from_prt (prt) result (c_prt)
        type(c_prt_t) :: c_prt
        type(prt_t), intent(in) :: prt
        c_prt%type = prt%type
        c_prt%pdg = prt%pdg
        if (prt%polarized) then
            c_prt%polarized = 1
        else
            c_prt%polarized = 0
        end if
    end function

```



```

c_prt%h = prt%h
c_prt%pe = energy (prt%p)
c_prt%px = vector4_get_component (prt%p, 1)
c_prt%py = vector4_get_component (prt%p, 2)
c_prt%pz = vector4_get_component (prt%p, 3)
c_prt%p2 = prt%p2
end function c_prt_from_prt

```

## Output

*<Subevents: public>+≡*

```
public :: prt_write
```

*<Subevents: procedures>+≡*

```

subroutine prt_write (prt, unit)
  type(prt_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, "(1x,A)", advance="no") "prt("
  select case (prt%type)
    case (PRT_UNDEFINED); write (u, "('?')", advance="no")
    case (PRT_BEAM); write (u, "('b:~')", advance="no")
    case (PRT_INCOMING); write (u, "('i:~')", advance="no")
    case (PRT_OUTGOING); write (u, "('o:~')", advance="no")
    case (PRT_COMPOSITE); write (u, "('c:~')", advance="no")
  end select
  select case (prt%type)
    case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
      if (prt%polarized) then
        write (u, "(I0,'/',I0,'|')", advance="no") prt%pdg, prt%h
      else
        write (u, "(I0,'|')", advance="no") prt%pdg
      end if
    end select
  select case (prt%type)
    case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING, PRT_COMPOSITE)
      write (u, "(ES14.7,';',ES14.7,';',ES14.7,';',ES14.7)", advance="no") &
        array_from_vector4 (prt%p)
      write (u, "('|',ES19.12)", advance="no") prt%p2
    end select
  if (allocated (prt%src)) then
    write (u, "('|')", advance="no")
    do i = 1, size (prt%src)
      write (u, "(1x,I0)", advance="no") prt%src(i)
    end do
  end if
  write (u, "(A)" ) " "
end subroutine prt_write

```

## Tools

Two particles match if their `src` arrays are the same.

```
<Subevents: interfaces>+≡
  interface operator(.match.)
    module procedure prt_match
  end interface

<Subevents: procedures>+≡
  elemental function prt_match (prt1, prt2) result (match)
    logical :: match
    type(prt_t), intent(in) :: prt1, prt2
    if (size (prt1%src) == size (prt2%src)) then
      match = all (prt1%src == prt2%src)
    else
      match = .false.
    end if
  end function prt_match
```

The combine operation makes a pseudoparticle whose momentum is the result of adding (the momenta of) the pair of input particles. We trace the particles from which a particle is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

The result is intent(inout), so it will not be initialized when the subroutine is entered.

```
<Subevents: procedures>+≡
  subroutine prt_combine (prt, prt_in1, prt_in2, ok)
    type(prt_t), intent(inout) :: prt
    type(prt_t), intent(in) :: prt_in1, prt_in2
    logical :: ok
    integer, dimension(:), allocatable :: src
    call combine_index_lists (src, prt_in1%src, prt_in2%src)
    ok = allocated (src)
    if (ok) call prt_init_composite (prt, prt_in1%p + prt_in2%p, src)
  end subroutine prt_combine
```

This variant does not produce the combined particle, it just checks whether the combination is valid (no common `src` entry).

```
<Subevents: public>+≡
  public :: are_disjoint

<Subevents: procedures>+≡
  function are_disjoint (prt_in1, prt_in2) result (flag)
    logical :: flag
    type(prt_t), intent(in) :: prt_in1, prt_in2
    flag = index_lists_are_disjoint (prt_in1%src, prt_in2%src)
  end function are_disjoint
```

`src` Lists with length > 1 are built by a `combine` operation which merges the lists in a sorted manner. If the result would have a duplicate entry, it is discarded, and the result is unallocated.

*(Subevents: procedures)+≡*

```
subroutine combine_index_lists (res, src1, src2)
  integer, dimension(:), intent(in) :: src1, src2
  integer, dimension(:), allocatable :: res
  integer :: i1, i2, i
  allocate (res (size (src1) + size (src2)))
  i1 = 1
  i2 = 1
  LOOP: do i = 1, size (res)
    if (src1(i1) < src2(i2)) then
      res(i) = src1(i1); i1 = i1 + 1
      if (i1 > size (src1)) then
        res(i+1:) = src2(i2:)
        exit LOOP
      end if
    else if (src1(i1) > src2(i2)) then
      res(i) = src2(i2); i2 = i2 + 1
      if (i2 > size (src2)) then
        res(i+1:) = src1(i1:)
        exit LOOP
      end if
    else
      deallocate (res)
      exit LOOP
    end if
  end do LOOP
end subroutine combine_index_lists
```

This function is similar, but it does not actually merge the list, it just checks whether they are disjoint (no common `src` entry).

*(Subevents: procedures)+≡*

```
function index_lists_are_disjoint (src1, src2) result (flag)
  logical :: flag
  integer, dimension(:), intent(in) :: src1, src2
  integer :: i1, i2, i
  flag = .true.
  i1 = 1
  i2 = 1
  LOOP: do i = 1, size (src1) + size (src2)
    if (src1(i1) < src2(i2)) then
      i1 = i1 + 1
      if (i1 > size (src1)) then
        exit LOOP
      end if
    else if (src1(i1) > src2(i2)) then
      i2 = i2 + 1
      if (i2 > size (src2)) then
        exit LOOP
      end if
    else
      exit LOOP
    end if
  end do LOOP
end function index_lists_are_disjoint
```

```

        flag = .false.
        exit LOOP
    end if
end do LOOP
end function index_lists_are_disjoint

```

### 5.3.2 C-compatible particle type

#### 5.3.3 subevents

Particles are collected in subevents. This type is implemented as a dynamically allocated array, which need not be completely filled. The value `n_active` determines the number of meaningful entries.

##### Type definition

```

<Subevents: public>+≡
    public :: subevt_t

<Subevents: types>+≡
    type :: subevt_t
        private
        integer :: n_active = 0
        type(prt_t), dimension(:), allocatable :: prt
    contains
        <Subevents: subevt: TBP>
    end type subevt_t

```

Initialize, allocating with size zero (default) or given size. The number of contained particles is set equal to the size.

```

<Subevents: public>+≡
    public :: subevt_init

<Subevents: procedures>+≡
    subroutine subevt_init (subevt, n_active)
        type(subevt_t), intent(out) :: subevt
        integer, intent(in), optional :: n_active
        if (present (n_active)) subevt%n_active = n_active
        allocate (subevt%prt (subevt%n_active))
    end subroutine subevt_init

```

(Re-)allocate the subevent with some given size. If the size is greater than the previous one, do a real reallocation. Otherwise, just reset the recorded size. Contents are untouched, but become invalid.

```

<Subevents: public>+≡
    public :: subevt_reset

```

```

<Subevents: procedures>+≡
  subroutine subevt_reset (subevt, n_active)
    type(subevt_t), intent(inout) :: subevt
    integer, intent(in) :: n_active
    subevt%n_active = n_active
    if (subevt%n_active > size (subevt%prt)) then
      deallocate (subevt%prt)
      allocate (subevt%prt (subevt%n_active))
    end if
  end subroutine subevt_reset

```

Output. No prefix for the headline 'subevt', because this will usually be printed appending to a previous line.

```

<Subevents: public>+≡
  public :: subevt_write

<Subevents: subevt: TBP>≡
  procedure :: write => subevt_write

<Subevents: procedures>+≡
  subroutine subevt_write (object, unit, prefix)
    class(subevt_t), intent(in) :: object
    integer, intent(in), optional :: unit
    character(*), intent(in), optional :: prefix
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A)") "subevent:"
    do i = 1, object%n_active
      if (present (prefix)) write (u, "(A)", advance="no") prefix
      write (u, "(1x,I0)", advance="no") i
      call prt_write (object%prt(i), unit)
    end do
  end subroutine subevt_write

```

Defined assignment: transfer only meaningful entries. This is a deep copy (as would be default assignment).

```

<Subevents: interfaces>+≡
  interface assignment(=)
    module procedure subevt_assign
  end interface

<Subevents: procedures>+≡
  subroutine subevt_assign (subevt, subevt_in)
    type(subevt_t), intent(inout) :: subevt
    type(subevt_t), intent(in) :: subevt_in
    if (.not. allocated (subevt%prt)) then
      call subevt_init (subevt, subevt_in%n_active)
    else
      call subevt_reset (subevt, subevt_in%n_active)
    end if
    subevt%prt(:subevt%n_active) = subevt_in%prt(:subevt%n_active)
  end subroutine subevt_assign

```

## Fill contents

Store incoming/outgoing particles which are completely defined.

*<Subevents: public>+≡*

```
public :: subevt_set_beam
public :: subevt_set_incoming
public :: subevt_set_outgoing
public :: subevt_set_composite
```

*<Subevents: procedures>+≡*

```
subroutine subevt_set_beam (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
    call prt_init_beam (subevt%prt(i), pdg, p, p2, src)
  else
    call prt_init_beam (subevt%prt(i), pdg, p, p2, (/ i /))
  end if
end subroutine subevt_set_beam

subroutine subevt_set_incoming (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
    call prt_init_incoming (subevt%prt(i), pdg, p, p2, src)
  else
    call prt_init_incoming (subevt%prt(i), pdg, p, p2, (/ i /))
  end if
end subroutine subevt_set_incoming

subroutine subevt_set_outgoing (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
    call prt_init_outgoing (subevt%prt(i), pdg, p, p2, src)
  else
    call prt_init_outgoing (subevt%prt(i), pdg, p, p2, (/ i /))
  end if
end subroutine subevt_set_outgoing

subroutine subevt_set_composite (subevt, i, p, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
```

```

    type(vector4_t), intent(in) :: p
    integer, dimension(:), intent(in) :: src
    call prt_init_composite (subvt%prt(i), p, src)
end subroutine subvt_set_composite

```

Separately assign flavors, simultaneously for all incoming/outgoing particles.

```

<Subevents: public>+≡
    public :: subvt_set_pdg_beam
    public :: subvt_set_pdg_incoming
    public :: subvt_set_pdg_outgoing

<Subevents: procedures>+≡
    subroutine subvt_set_pdg_beam (subvt, pdg)
        type(subvt_t), intent(inout) :: subvt
        integer, dimension(:), intent(in) :: pdg
        integer :: i, j
        j = 1
        do i = 1, subvt%n_active
            if (subvt%prt(i)%type == PRT_BEAM) then
                call prt_set_pdg (subvt%prt(i), pdg(j))
                j = j + 1
                if (j > size (pdg)) exit
            end if
        end do
    end subroutine subvt_set_pdg_beam

    subroutine subvt_set_pdg_incoming (subvt, pdg)
        type(subvt_t), intent(inout) :: subvt
        integer, dimension(:), intent(in) :: pdg
        integer :: i, j
        j = 1
        do i = 1, subvt%n_active
            if (subvt%prt(i)%type == PRT_INCOMING) then
                call prt_set_pdg (subvt%prt(i), pdg(j))
                j = j + 1
                if (j > size (pdg)) exit
            end if
        end do
    end subroutine subvt_set_pdg_incoming

    subroutine subvt_set_pdg_outgoing (subvt, pdg)
        type(subvt_t), intent(inout) :: subvt
        integer, dimension(:), intent(in) :: pdg
        integer :: i, j
        j = 1
        do i = 1, subvt%n_active
            if (subvt%prt(i)%type == PRT_OUTGOING) then
                call prt_set_pdg (subvt%prt(i), pdg(j))
                j = j + 1
                if (j > size (pdg)) exit
            end if
        end do
    end subroutine subvt_set_pdg_outgoing

```

Separately assign momenta, simultaneously for all incoming/outgoing particles.

```

<Subevents: public>+≡
  public :: subevt_set_p_beam
  public :: subevt_set_p_incoming
  public :: subevt_set_p_outgoing

<Subevents: procedures>+≡
  subroutine subevt_set_p_beam (subevt, p)
    type(subevt_t), intent(inout) :: subevt
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i, j
    j = 1
    do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_BEAM) then
        call prt_set_p (subevt%prt(i), p(j))
        j = j + 1
        if (j > size (p)) exit
      end if
    end do
  end subroutine subevt_set_p_beam

  subroutine subevt_set_p_incoming (subevt, p)
    type(subevt_t), intent(inout) :: subevt
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i, j
    j = 1
    do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_INCOMING) then
        call prt_set_p (subevt%prt(i), p(j))
        j = j + 1
        if (j > size (p)) exit
      end if
    end do
  end subroutine subevt_set_p_incoming

  subroutine subevt_set_p_outgoing (subevt, p)
    type(subevt_t), intent(inout) :: subevt
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i, j
    j = 1
    do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_OUTGOING) then
        call prt_set_p (subevt%prt(i), p(j))
        j = j + 1
        if (j > size (p)) exit
      end if
    end do
  end subroutine subevt_set_p_outgoing

```

Separately assign the squared invariant mass, simultaneously for all incoming/outgoing particles.

```

<Subevents: public>+≡
  public :: subevt_set_p2_beam
  public :: subevt_set_p2_incoming

```



```

public :: subevt_set_p2_outgoing
<Subevents: procedures>+≡
subroutine subevt_set_p2_beam (subevt, p2)
  type(subevt_t), intent(inout) :: subevt
  real(default), dimension(:), intent(in) :: p2
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
    if (subevt%prt(i)%type == PRT_BEAM) then
      call prt_set_p2 (subevt%prt(i), p2(j))
      j = j + 1
      if (j > size (p2)) exit
    end if
  end do
end subroutine subevt_set_p2_beam

subroutine subevt_set_p2_incoming (subevt, p2)
  type(subevt_t), intent(inout) :: subevt
  real(default), dimension(:), intent(in) :: p2
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
    if (subevt%prt(i)%type == PRT_INCOMING) then
      call prt_set_p2 (subevt%prt(i), p2(j))
      j = j + 1
      if (j > size (p2)) exit
    end if
  end do
end subroutine subevt_set_p2_incoming

subroutine subevt_set_p2_outgoing (subevt, p2)
  type(subevt_t), intent(inout) :: subevt
  real(default), dimension(:), intent(in) :: p2
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
    if (subevt%prt(i)%type == PRT_OUTGOING) then
      call prt_set_p2 (subevt%prt(i), p2(j))
      j = j + 1
      if (j > size (p2)) exit
    end if
  end do
end subroutine subevt_set_p2_outgoing

```

Set polarization for an entry

```

<Subevents: public>+≡
public :: subevt_polarize
<Subevents: procedures>+≡
subroutine subevt_polarize (subevt, i, h)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i, h
  call prt_polarize (subevt%prt(i), h)
end subroutine subevt_polarize

```

## Accessing contents

Return true if the subevent has entries.

```
<Subevents: public>+≡  
    public :: subevt_is_nonempty  
  
<Subevents: procedures>+≡  
    function subevt_is_nonempty (subevt) result (flag)  
        logical :: flag  
        type(subevt_t), intent(in) :: subevt  
        flag = subevt%n_active /= 0  
    end function subevt_is_nonempty
```

Return the number of entries

```
<Subevents: public>+≡  
    public :: subevt_get_length  
  
<Subevents: procedures>+≡  
    function subevt_get_length (subevt) result (length)  
        integer :: length  
        type(subevt_t), intent(in) :: subevt  
        length = subevt%n_active  
    end function subevt_get_length
```

Return a specific particle. The index is not checked for validity.

```
<Subevents: public>+≡  
    public :: subevt_get_prt  
  
<Subevents: procedures>+≡  
    function subevt_get_prt (subevt, i) result (prt)  
        type(prt_t) :: prt  
        type(subevt_t), intent(in) :: subevt  
        integer, intent(in) :: i  
        prt = subevt%prt(i)  
    end function subevt_get_prt
```

Return the partonic energy squared. We take the particles with flag PRT\_INCOMING and compute their total invariant mass.

```
<Subevents: public>+≡  
    public :: subevt_get_sqrts_hat  
  
<Subevents: procedures>+≡  
    function subevt_get_sqrts_hat (subevt) result (sqrts_hat)  
        type(subevt_t), intent(in) :: subevt  
        real(default) :: sqrts_hat  
        type(vector4_t) :: p  
        integer :: i  
        do i = 1, subevt%n_active  
            if (subevt%prt(i)%type == PRT_INCOMING) then  
                p = p + prt_get_momentum (subevt%prt(i))  
            end if  
        end do  
    end function subevt_get_sqrts_hat
```

```

end do
sqrtshat = p ** 1
end function subevt_get_sqrtshat

```

*<Subevents: interfaces>+≡*

```

interface c_prt
  module procedure c_prt_from_subevt
  module procedure c_prt_array_from_subevt
end interface

```

*<Subevents: procedures>+≡*

```

function c_prt_from_subevt (subevt, i) result (c_prt)
  type(c_prt_t) :: c_prt
  type(subevt_t), intent(in) :: subevt
  integer, intent(in) :: i
  c_prt = c_prt_from_prt (subevt%prt(i))
end function c_prt_from_subevt

function c_prt_array_from_subevt (subevt) result (c_prt_array)
  type(subevt_t), intent(in) :: subevt
  type(c_prt_t), dimension(subevt%n_active) :: c_prt_array
  c_prt_array = c_prt_from_prt (subevt%prt(1:subevt%n_active))
end function c_prt_array_from_subevt

```

## Operations with subevents

The join operation joins two subevents. When appending the elements of the second list, we check for each particle whether it is already in the first list. If yes, it is discarded. The result list should be initialized already.

If a mask is present, it refers to the second subevent. Particles where the mask is not set are discarded.

*<Subevents: public>+≡*

```

public :: subevt_join

```

*<Subevents: procedures>+≡*

```

subroutine subevt_join (subevt, pl1, pl2, mask2)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1, pl2
  logical, dimension(:), intent(in), optional :: mask2
  integer :: n1, n2, i, n
  n1 = pl1%n_active
  n2 = pl2%n_active
  call subevt_reset (subevt, n1 + n2)
  subevt%prt(:n1) = pl1%prt(:n1)
  n = n1
  if (present (mask2)) then
    do i = 1, pl2%n_active
      if (mask2(i)) then
        if (disjoint (i)) then
          n = n + 1
          subevt%prt(n) = pl2%prt(i)
        end if
      end if
    end do
  end if

```

```

        end if
    end do
else
    do i = 1, pl2%n_active
        if (disjoint (i)) then
            n = n + 1
            subevt%prt(n) = pl2%prt(i)
        end if
    end do
end if
subevt%n_active = n
contains
function disjoint (i) result (flag)
    integer, intent(in) :: i
    logical :: flag
    integer :: j
    do j = 1, pl1%n_active
        if (.not. are_disjoint (pl1%prt(j), pl2%prt(i))) then
            flag = .false.
            return
        end if
    end do
    flag = .true.
end function disjoint
end subroutine subevt_join

```

The combine operation makes a subevent whose entries are the result of adding (the momenta of) each pair of particles in the input lists. We trace the particles from which a particle is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

*<Subevents: public>+≡*

```
public :: subevt_combine
```

*<Subevents: procedures>+≡*

```

subroutine subevt_combine (subevt, pl1, pl2, mask12)
    type(subevt_t), intent(inout) :: subevt
    type(subevt_t), intent(in) :: pl1, pl2
    logical, dimension(:,,:), intent(in), optional :: mask12
    integer :: n1, n2, i1, i2, n, j
    logical :: ok
    n1 = pl1%n_active
    n2 = pl2%n_active
    call subevt_reset (subevt, n1 * n2)
    n = 1
    do i1 = 1, n1
        do i2 = 1, n2
            if (present (mask12)) then
                ok = mask12(i1,i2)
            else

```

```

        ok = .true.
    end if
    if (ok) call prt_combine &
        (subevt%prt(n), pl1%prt(i1), pl2%prt(i2), ok)
    if (ok) then
        CHECK_DOUBLES: do j = 1, n - 1
            if (subevt%prt(n) .match. subevt%prt(j)) then
                ok = .false.; exit CHECK_DOUBLES
            end if
        end do CHECK_DOUBLES
        if (ok) n = n + 1
    end if
end do
subevt%n_active = n - 1
end subroutine subevt_combine

```

The collect operation makes a single-entry subevent which results from combining (the momenta of) all particles in the input list. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

(Thus, if the input particles are already composite, there is some chance that the result depends on the order of the input list and is not as expected. This situation should be avoided.)

```

<Subevents: public>+≡
    public :: subevt_collect

<Subevents: procedures>+≡
    subroutine subevt_collect (subevt, pl1, mask1)
        type(subevt_t), intent(inout) :: subevt
        type(subevt_t), intent(in) :: pl1
        logical, dimension(:), intent(in) :: mask1
        type(prt_t) :: prt
        integer :: i
        logical :: ok
        call subevt_reset (subevt, 1)
        subevt%n_active = 0
        do i = 1, pl1%n_active
            if (mask1(i)) then
                if (subevt%n_active == 0) then
                    subevt%n_active = 1
                    subevt%prt(1) = pl1%prt(i)
                else
                    call prt_combine (prt, subevt%prt(1), pl1%prt(i), ok)
                    if (ok) subevt%prt(1) = prt
                end if
            end if
        end do
    end subroutine subevt_collect

```

Return a list of all particles for which the mask is true.

```

<Subevents: public>+≡

```

```

public :: subevt_select
<Subevents: procedures>+≡
subroutine subevt_select (subevt, pl, mask1)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  logical, dimension(:), intent(in) :: mask1
  integer :: i, n
  call subevt_reset (subevt, pl%n_active)
  n = 0
  do i = 1, pl%n_active
    if (mask1(i)) then
      n = n + 1
      subevt%prt(n) = pl%prt(i)
    end if
  end do
  subevt%n_active = n
end subroutine subevt_select

```

Return a subevent which consists of the single particle with specified `index`. If `index` is negative, count from the end. If it is out of bounds, return an empty list.

```

<Subevents: public>+≡
public :: subevt_extract
<Subevents: procedures>+≡
subroutine subevt_extract (subevt, pl, index)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  integer, intent(in) :: index
  if (index > 0) then
    if (index <= pl%n_active) then
      call subevt_reset (subevt, 1)
      subevt%prt(1) = pl%prt(index)
    else
      call subevt_reset (subevt, 0)
    end if
  else if (index < 0) then
    if (abs (index) <= pl%n_active) then
      call subevt_reset (subevt, 1)
      subevt%prt(1) = pl%prt(pl%n_active + 1 + index)
    else
      call subevt_reset (subevt, 0)
    end if
  else
    call subevt_reset (subevt, 0)
  end if
end subroutine subevt_extract

```

Return the list of particles sorted according to increasing values of the provided integer or real array. If no array is given, sort by PDG value.

```

<Subevents: public>+≡
public :: subevt_sort

```

```

<Subevents: interfaces>+≡
  interface subevt_sort
    module procedure subevt_sort_pdg
    module procedure subevt_sort_int
    module procedure subevt_sort_real
  end interface

<Subevents: procedures>+≡
  subroutine subevt_sort_pdg (subevt, pl)
    type(subevt_t), intent(inout) :: subevt
    type(subevt_t), intent(in) :: pl
    integer :: n
    n = subevt%n_active
    call subevt_sort_int (subevt, pl, abs (3 * subevt%prt(:n)%pdg - 1))
  end subroutine subevt_sort_pdg

  subroutine subevt_sort_int (subevt, pl, ival)
    type(subevt_t), intent(inout) :: subevt
    type(subevt_t), intent(in) :: pl
    integer, dimension(:), intent(in) :: ival
    call subevt_reset (subevt, pl%n_active)
    subevt%n_active = pl%n_active
    subevt%prt = pl%prt( order (ival) )
  end subroutine subevt_sort_int

  subroutine subevt_sort_real (subevt, pl, rval)
    type(subevt_t), intent(inout) :: subevt
    type(subevt_t), intent(in) :: pl
    real(default), dimension(:), intent(in) :: rval
    call subevt_reset (subevt, pl%n_active)
    subevt%n_active = pl%n_active
    subevt%prt = pl%prt( order (rval) )
  end subroutine subevt_sort_real

```

Return the list of particles which have any of the specified PDG codes (and optionally particle type: beam, incoming, outgoing).

The `pack` command was buggy in some gfortran versions, therefore it is unrolled. The unrolled version may be more efficient, actually.

```

<Subevents: public>+≡
  public :: subevt_select_pdg_code

<Subevents: procedures>+≡
  subroutine subevt_select_pdg_code (subevt, aval, subevt_in, prt_type)
    type(subevt_t), intent(inout) :: subevt
    type(pdg_array_t), intent(in) :: aval
    type(subevt_t), intent(in) :: subevt_in
    integer, intent(in), optional :: prt_type
    integer :: n_active, n_match
    logical, dimension(:), allocatable :: mask
    integer :: i, j
    n_active = subevt_in%n_active
    allocate (mask (n_active))
    forall (i = 1:n_active) &

```

```

        mask(i) = aval .match. subevt_in%prt(i)%pdg
    if (present (prt_type)) &
        mask = mask .and. subevt_in%prt(:n_active)%type == prt_type
    n_match = count (mask)
    call subevt_reset (subevt, n_match)
!    subevt%prt(:n_match) = pack (subevt_in%prt(:n_active), mask)
    j = 0
    do i = 1, n_active
        if (mask(i)) then
            j = j + 1
            subevt%prt(j) = subevt_in%prt(i)
        end if
    end do
end subroutine subevt_select_pdg_code

```

### 5.3.4 Eliminate numerical noise

This is useful for testing purposes: set entries to zero that are smaller in absolute values than a given tolerance parameter.

```

<Subevents: public>+≡
    public :: pacify

<Subevents: interfaces>+≡
    interface pacify
        module procedure pacify_prt
        module procedure pacify_subevt
    end interface pacify

<Subevents: procedures>+≡
    subroutine pacify_prt (prt)
        class(prt_t), intent(inout) :: prt
        real(default) :: e
        e = epsilon (1._default) * energy (prt%p)
        call pacify (prt%p, 10 * e)
        call pacify (prt%p2, 1e4 * e)
    end subroutine pacify_prt

    subroutine pacify_subevt (subevt)
        class(subevt_t), intent(inout) :: subevt
        integer :: i
        do i = 1, subevt%n_active
            call pacify (subevt%prt(i))
        end do
    end subroutine pacify_subevt

```

## 5.4 User Code Interface

Here we collect interface code that enables the user to inject his own code into the WHIZARD workflow. The code uses data types defined above, and is



referenced in the `expressions` module.

```
(user_code_interface.f90)≡
  <File header>

  module user_code_interface

    use iso_c_binding !NODEP!
    <Use kinds>
    <Use strings>
    use diagnostics !NODEP!
    use c_particles !NODEP!
    use os_interface

    <Standard module head>

    <User Code: public>

    <User Code: variables>

    <User Code: interfaces>

    contains

    <User Code: procedures>

  end module user_code_interface
```

### 5.4.1 User Code Management

This data structure globally holds the filehandle of the user-code library:

```
<User Code: public>≡
  public :: has_user_lib

<User Code: variables>≡
  type(dlaccess_t), save :: user_lib_handle
  logical, save :: has_user_lib = .false.
  type(string_t), save :: user
```

Compile, link and load user code files. Dlopen all user-provided libraries, included the one just compiled (if any).

By default, we are looking for a library `user.so`. If this is not present, try `user.f90` and compile it. This can be overridden.

In detail: First, compile all sources explicitly specified on the command line. Then collect all libraries specified on the command line, including `user.so` if it was generated. If there is still no code, check for an existing `user.f90` and compile this. Link everything into a `user.la` libtool library. When done, dlopen all libraries that we have so far.

```
<User Code: public>+≡
  public :: user_code_init
```

```

{User Code: procedures}≡
subroutine user_code_init (user_src, user_lib, user_target, rebuild, os_data)
  type(string_t), dimension(:), intent(in) :: user_src, user_lib
  type(string_t), intent(in) :: user_target
  logical, intent(in) :: rebuild
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: user_src_file, user_obj_files, user_lib_file
  logical :: exist
  type(c_funptr) :: fptr
  integer :: i
  call msg_message ("Initializing user code")
  user = user_target; if (user == "") user = "user"
  user_obj_files = ""
  inquire (file = char (user) // ".la", exist = exist)
  if (rebuild .or. .not. exist) then
    do i = 1, size (user_src)
      user_src_file = user_src(i) // os_data%fc_src_ext
      inquire (file = char (user_src_file), exist = exist)
      if (exist) then
        call msg_message ("Found user-code source '" &
          // char (user_src_file) // "'.")
        call compile_user_src (user_src_file, user_obj_files)
      else
        call msg_fatal ("User-code source '" // char (user_src_file) &
          // "' not found")
      end if
    end do
    do i = 1, size (user_lib)
      user_lib_file = user_lib(i) // ".la"
      inquire (file = char (user_lib_file), exist = exist)
      if (exist) then
        call msg_message ("Found user-code library '" &
          // char (user_lib_file) // "'.")
      else
        user_lib_file = user_lib(i) // os_data%shlib_ext
        inquire (file = char (user_lib_file), exist = exist)
        if (exist) then
          call msg_message ("Found user-code library '" &
            // char (user_lib_file) // "'.")
        else
          call msg_fatal ("User-code library '" // char (user_lib(i)) &
            // "' not found")
        end if
      end if
    end do
    user_obj_files = user_obj_files // " " // user_lib_file
  end do
  if (user_obj_files == "") then
    user_src_file = user // os_data%fc_src_ext
    inquire (file = char (user_src_file), exist = exist)
    if (exist) then
      call msg_message ("Found user-code source '" &
        // char (user_src_file) // "'.")
      call compile_user_src (user_src_file, user_obj_files)
    else

```

```

        call msg_fatal ("User-code source '" // char (user_src_file) &
            // "' not found")
    end if
end if
if (user_obj_files /= "") then
    call link_user (char (user), user_obj_files)
end if
end if
call dlaccess_init &
    (user_lib_handle, var_str ("."), &
    user // os_data%shlib_ext, os_data)
if (dlaccess_has_error (user_lib_handle)) then
    call msg_error (char (dlaccess_get_error (user_lib_handle)))
    call msg_fatal ("Loading user code library '" // char (user) &
        // ".la' failed")
else
    call msg_message ("User code library '" // char (user) &
        // ".la' successfully loaded")
    has_user_lib = .true.
end if
contains
subroutine compile_user_src (user_src_file, user_obj_files)
    type(string_t), intent(in) :: user_src_file
    type(string_t), intent(inout) :: user_obj_files
    type(string_t) :: basename, ext
    logical :: exist
    basename = user_src_file
    call split (basename, ext, ".", back=.true.)
    if ( "." // ext == os_data%fc_src_ext) then
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
            call msg_message ("Compiling user code file '" &
                // char (user_src_file) // "'")
            call os_compile_shared (basename, os_data)
            user_obj_files = user_obj_files // " " // basename // ".lo"
        else
            call msg_error ("User code file '" // char (user_src_file) &
                // "' not found.")
        end if
    else
        call msg_error ("User code file '" // char (user_src_file) &
            // "' should have file extension '" &
            // char (os_data%fc_src_ext) // "'")
    end if
end subroutine compile_user_src
subroutine link_user (user_lib, user_obj_files)
    character(*), intent(in) :: user_lib
    type(string_t), intent(in) :: user_obj_files
    call msg_message ("Linking user code library '" &
        // user_lib // char (os_data%shlib_ext) // "'")
    call os_link_shared (user_obj_files, var_str (user_lib), os_data)
end subroutine link_user
end subroutine user_code_init

```

Unload all user-code libraries.

```
<User Code: public>+≡
    public :: user_code_final

<User Code: procedures>+≡
    subroutine user_code_final ()
        if (has_user_lib) then
            call dlaccess_final (user_lib_handle)
            has_user_lib = .false.
        end if
    end subroutine user_code_final
```

Try to load the possible user-defined procedures from the dlopened libraries. If a procedure is not found, do nothing.

```
<User Code: public>+≡
    public :: user_code_find_proc

<User Code: procedures>+≡
    function user_code_find_proc (name) result (fptr)
        type(string_t), intent(in) :: name
        type(c_funptr) :: fptr
        integer :: i
        fptr = c_null_funptr
        fptr = libmanager_get_c_funptr (char (user), char (name))
        if (.not. c_associated (fptr)) then
            if (has_user_lib) then
                fptr = dlaccess_get_c_funptr (user_lib_handle, name)
                if (.not. c_associated (fptr)) then
                    call msg_fatal ("User procedure '" // char (name) // "' not found")
                end if
            else
                call msg_fatal ("User procedure '" // char (name) &
                    // "' called without user library (missing -u flag?)")
            end if
        end if
    end function user_code_find_proc
```

### 5.4.2 Libmanager interface

This is the interface to the libmanager library. The libmanager is generated when a static executable is built. The definitions are in Sec. 14.4.

```
<User Code: interfaces>≡
    <Libmanager: interfaces>

<Libmanager: interfaces>≡
    interface
        function libmanager_get_n_libs () result (n)
            integer :: n
        end function libmanager_get_n_libs
    end interface
```

```

<Libmanager: interfaces>+≡
interface
  function libmanager_get_libname (i) result (name)
    use iso_varying_string, string_t => varying_string !NODEP!
    type(string_t) :: name
    integer, intent(in) :: i
  end function libmanager_get_libname
end interface

```

```

<Libmanager: interfaces>+≡
interface
  function libmanager_get_c_funptr (libname, fname) result (c_fptr)
    use iso_c_binding !NODEP!
    type(c_funptr) :: c_fptr
    character(*), intent(in) :: libname, fname
  end function libmanager_get_c_funptr
end interface

```

### 5.4.3 Interfaces for user-defined functions

The following functions represent user-defined real observables. There may be one or two particles as argument, the result is a real value.

```

<User Code: public>+≡
public :: user_obs_int_unary
public :: user_obs_int_binary
public :: user_obs_real_unary
public :: user_obs_real_binary

<User Code: interfaces>+≡
abstract interface
  function user_obs_int_unary (prt1) result (ival) bind(C)
    use iso_c_binding !NODEP!
    use c_particles !NODEP!
    type(c_prt_t), intent(in) :: prt1
    integer(c_int) :: ival
  end function user_obs_int_unary
end interface

abstract interface
  function user_obs_int_binary (prt1, prt2) result (ival) bind(C)
    use iso_c_binding !NODEP!
    use c_particles !NODEP!
    type(c_prt_t), intent(in) :: prt1, prt2
    integer(c_int) :: ival
  end function user_obs_int_binary
end interface

abstract interface
  function user_obs_real_unary (prt1) result (rval) bind(C)
    use iso_c_binding !NODEP!
    use c_particles !NODEP!
    type(c_prt_t), intent(in) :: prt1

```

```

        real(c_double) :: rval
    end function user_obs_real_unary
end interface

abstract interface
    function user_obs_real_binary (prt1, prt2) result (rval) bind(C)
        use iso_c_binding !NODEP!
        use c_particles !NODEP!
        type(c_prt_t), intent(in) :: prt1, prt2
        real(c_double) :: rval
    end function user_obs_real_binary
end interface

```

The following function takes an array of C-compatible particles and return a single value. The particle array represents a subevent. For C interoperability, we have to use an assumed-size array, hence the array size has to be transferred explicitly.

The cut function returns an `int`, which we should interpret as a logical value (`nonzero=true`).

```

<User Code: public>+≡
    public :: user_cut_fun

<User Code: interfaces>+≡
    abstract interface
        function user_cut_fun (prt, n_prt) result (iflag) bind(C)
            use iso_c_binding !NODEP!
            use c_particles !NODEP!
            type(c_prt_t), dimension(*), intent(in) :: prt
            integer(c_int), intent(in) :: n_prt
            integer(c_int) :: iflag
        end function user_cut_fun
    end interface

```

The event-shape function returns a real value.

```

<User Code: public>+≡
    public :: user_event_shape_fun

<User Code: interfaces>+≡
    abstract interface
        function user_event_shape_fun (prt, n_prt) result (rval) bind(C)
            use iso_c_binding !NODEP!
            use c_particles !NODEP!
            type(c_prt_t), dimension(*), intent(in) :: prt
            integer(c_int), intent(in) :: n_prt
            real(c_double) :: rval
        end function user_event_shape_fun
    end interface

```

#### 5.4.4 Interfaces for user-defined interactions

The following procedure interfaces pertain to user-defined interactions, e.g., spectra or structure functions.

This subroutine retrieves the basic information for setting up the interaction and event generation. All parameters are `intent(inout)`, so we can provide default values. `n_in` and `n_out` are the number of incoming and outgoing particles, respectively. `n_states` is the total number of distinct states that should be generated (counting all states of the incoming particles). `n_col` is the maximal number of color entries a particle can have. `n_dim` is the number of input parameters, i.e., integration dimensions, that the structure function call requires for computing kinematics and matrix elements. `n_var` is the number of variables (e.g., momentum fractions) that the structure function call has to transfer from the kinematics to the dynamics evaluation.

```

<User Code: public>+≡
  public :: user_int_info

<User Code: interfaces>+≡
  abstract interface
    subroutine user_int_info (n_in, n_out, n_states, n_col, n_dim, n_var) &
      bind(C)
      use iso_c_binding !NODEP!
      integer(c_int), intent(inout) :: n_in, n_out, n_states, n_col
      integer(c_int), intent(inout) :: n_dim, n_var
    end subroutine user_int_info
  end interface

```

This subroutine retrieves the settings for the quantum number mask of a given particle index in the interaction. A nonzero value indicates that the corresponding quantum number is to be ignored. The lock index is the index of a particle that the current particle is related to. The relation applies if quantum numbers of one of the locked particles are summed over. (This is intended for helicity.)

```

<User Code: public>+≡
  public :: user_int_mask

<User Code: interfaces>+≡
  abstract interface
    subroutine user_int_mask (i_prt, m_flv, m_hel, m_col, i_lock) bind(C)
      use iso_c_binding !NODEP!
      integer(c_int), intent(in) :: i_prt
      integer(c_int), intent(inout) :: m_flv, m_hel, m_col, i_lock
    end subroutine user_int_mask
  end interface

```

This subroutine retrieves the quantum numbers for the particle index `i_prt` in state `i_state`. The `flv` value is a PDG code. The `hel` value is an integer helicity (twice the helicity for fermions). The `col` array is an array which has at most `n_col` entries (see above). All parameters are `intent(inout)` since default values exist. In particular, if a mask entry is set by the previous procedure, the corresponding quantum number is ignored anyway.

```

<User Code: public>+≡
  public :: user_int_state

<User Code: interfaces>+≡
  abstract interface
    subroutine user_int_state (i_state, i_prt, flv, hel, col) bind(C)

```

```

        use iso_c_binding !NODEP!
        integer(c_int), intent(in) :: i_state, i_prt
        integer(c_int), intent(inout) :: flv, hel
        integer(c_int), dimension(*), intent(inout) :: col
    end subroutine user_int_state
end interface

```

This subroutine takes an array of particle objects with array length `n_in` and an array of input parameters between 0 and 1 with array length `n_dim`. It returns an array of particle objects with array length `n_out`. In addition, it returns an array of internal variables (e.g., momentum fractions, Jacobian) with array length `n_var` that is used by the following subroutine for evaluating the dynamics, i.e., the matrix elements.

```

<User Code: public>+≡
    public :: user_int_kinematics

<User Code: interfaces>+≡
    abstract interface
        subroutine user_int_kinematics (prt_in, rval, prt_out, xval) bind(C)
            use iso_c_binding !NODEP!
            use c_particles !NODEP!
            type(c_prt_t), dimension(*), intent(in) :: prt_in
            real(c_double), dimension(*), intent(in) :: rval
            type(c_prt_t), dimension(*), intent(inout) :: prt_out
            real(c_double), dimension(*), intent(out) :: xval
        end subroutine user_int_kinematics
    end interface

```

This subroutine takes the array of variables (e.g., momentum fractions) with length `n_var` which has been generated by the previous subroutine and a real variable, the energy scale of the event. It returns an array of matrix-element values, one entry for each quantum state `n_states`. The ordering of matrix elements must correspond to the ordering of states.

```

<User Code: public>+≡
    public :: user_int_evaluate

<User Code: interfaces>+≡
    abstract interface
        subroutine user_int_evaluate (xval, scale, fval) bind(C)
            use iso_c_binding !NODEP!
            real(c_double), dimension(*), intent(in) :: xval
            real(c_double), intent(in) :: scale
            real(c_double), dimension(*), intent(out) :: fval
        end subroutine user_int_evaluate
    end interface

```

## 5.5 Variables

The user interface deals with variables that are handled similar to full-fledged programming languages. The system will add a lot of predefined variables



(model parameters, flags, etc.) that are accessible to the user by the same methods.

Variables can be of various type: logical (boolean/flag), integer, real (default precision), subevents (used in cut expressions), arrays of PDG codes (aliases for particles), strings. Furthermore, in cut expressions we have unary and binary observables, which are used like real parameters but behave like functions.

```

<variables.f90>≡
  <File header>

  module variables

    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use pdg_arrays
    use subevents

    <Standard module head>

    <Variables: public>

    <Variables: parameters>

    <Variables: types>

    <Variables: interfaces>

    contains

    <Variables: procedures>

  end module variables

```

### 5.5.1 Variable list entries

Variable (and constant) values can be of one of the following types:

```

<Variables: parameters>≡
  integer, parameter, public :: V_NONE = 0, V_LOG = 1, V_INT = 2, V_REAL = 3
  integer, parameter, public :: V_CMLPX = 4, V_SEV = 5, V_PDG = 6, V_STR = 7
  integer, parameter, public :: V_OBS1_INT = 11, V_OBS2_INT = 12
  integer, parameter, public :: V_OBS1_REAL = 21, V_OBS2_REAL = 22
  integer, parameter, public :: V_UOBS1_INT = 31, V_UOBS2_INT = 32
  integer, parameter, public :: V_UOBS1_REAL = 41, V_UOBS2_REAL = 42

```

#### The type

This is an entry in the variable list. It can be of any type; in each case only one value is allocated. It may be physically allocated upon creation, in which case

`is_allocated` is true, or it may contain just a pointer to a value somewhere else, in which case `is_allocated` is false.

The flag `is_defined` is set when the variable is given a value, even the undefined value. (Therefore it is distinct from `is_known`.) This matters for variable declaration in the SINDARIN language. The variable is set up in the compilation step and initially marked as defined, but after compilation all variables are set undefined. Each variable becomes defined when it is explicitly set. The difference matters in loops.

`is_locked` means that it cannot be given a value using the interface routines `var_list_set_XXX` below. It can only be initialized, or change automatically due to a side effect.

`is_copy` means that this is a local copy of a global variable. The copy has a pointer to the original, which can be used to restore a previous value.

`is_intrinsic` means that this variable is defined by the program, not by the user. Intrinsic variables cannot be (re)declared, but their values can be reset unless they are locked. `is_user_var` means that the variable has been declared by the user. It could be a new variable, or a local copy of an intrinsic variable.

The flag `is_known` is a pointer which parallels the use of the value pointer. For pointer variables, it is set if the value should point to a known value. For ordinary variables, it should be true.

The value is implemented as a set of alternative type-specific pointers. This emulates polymorphism, and it allows for actual pointer variables. Observable-type variables have function pointers as values, so they behave like macros. The functions make use of the particle objects accessible via the pointers `p1` and `p2`.

Finally, the `next` pointer indicates that we are making lists of variables. A more efficient implementation might switch to hashes or similar; the current implementation has  $O(N)$  lookup.

```

<Variables: public>≡
    public :: var_entry_t

<Variables: types>≡
    type :: var_entry_t
    private
    integer :: type = V_NONE
    type(string_t) :: name
    logical :: is_allocated = .false.
    logical :: is_defined = .false.
    logical :: is_locked = .false.
    logical :: is_copy = .false.
    type(var_entry_t), pointer :: original => null ()
    logical :: is_intrinsic = .false.
    logical :: is_user_var = .false.
    logical, pointer :: is_known => null ()
    logical,          pointer :: lval => null ()
    integer,          pointer :: ival => null ()
    real(default),    pointer :: rval => null ()
    complex(default), pointer :: cval => null ()
    type(subvt_t),    pointer :: pval => null ()
    type(pdg_array_t), pointer :: aval => null ()
    type(string_t),    pointer :: sval => null ()
    procedure(obs_unary_int), nopass, pointer :: obs1_int => null ()

```

```

    procedure(obs_unary_real), nopass, pointer :: obs1_real => null ()
    procedure(obs_binary_int), nopass, pointer :: obs2_int => null ()
    procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
    type(prt_t), pointer :: prt1 => null ()
    type(prt_t), pointer :: prt2 => null ()
    type(var_entry_t), pointer :: next => null ()
end type var_entry_t

```

## Interfaces for the observable functions

```

<Variables: public>+≡
    public :: obs_unary_int
    public :: obs_unary_real
    public :: obs_binary_int
    public :: obs_binary_real

<Variables: interfaces>≡
    abstract interface
        function obs_unary_int (prt1) result (ival)
            import
            integer :: ival
            type(prt_t), intent(in) :: prt1
        end function obs_unary_int
    end interface
    abstract interface
        function obs_unary_real (prt1) result (rval)
            import
            real(default) :: rval
            type(prt_t), intent(in) :: prt1
        end function obs_unary_real
    end interface
    abstract interface
        function obs_binary_int (prt1, prt2) result (ival)
            import
            integer :: ival
            type(prt_t), intent(in) :: prt1, prt2
        end function obs_binary_int
    end interface
    abstract interface
        function obs_binary_real (prt1, prt2) result (rval)
            import
            real(default) :: rval
            type(prt_t), intent(in) :: prt1, prt2
        end function obs_binary_real
    end interface

```

## Initialization

Initialize an entry, optionally with a physical value. We also allocate the `is_known` flag and set it if the value is set.

```

<Variables: public>+≡
    public :: var_entry_init_log

```

```

public :: var_entry_init_int
public :: var_entry_init_real
public :: var_entry_init_cmplx
public :: var_entry_init_pdg_array
public :: var_entry_init_subevt
public :: var_entry_init_string

```

*{Variables: procedures}*≡

```

subroutine var_entry_init_log (var, name, lval, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: lval
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_LOG
  allocate (var%lval, var%is_known)
  if (present (lval)) then
    var%lval = lval
    var%is_defined = .true.
    var%is_known = .true.
  else
    var%is_known = .false.
  end if
  if (present (intrinsic)) var%is_intrinsic = intrinsic
  if (present (user)) var%is_user_var = user
  var%is_allocated = .true.
end subroutine var_entry_init_log

```

```

subroutine var_entry_init_int (var, name, ival, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  integer, intent(in), optional :: ival
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_INT
  allocate (var%ival, var%is_known)
  if (present (ival)) then
    var%ival = ival
    var%is_defined = .true.
    var%is_known = .true.
  else
    var%is_known = .false.
  end if
  if (present (intrinsic)) var%is_intrinsic = intrinsic
  if (present (user)) var%is_user_var = user
  var%is_allocated = .true.
end subroutine var_entry_init_int

```

```

subroutine var_entry_init_real (var, name, rval, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  real(default), intent(in), optional :: rval
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_REAL

```

```

allocate (var%rval, var%is_known)
if (present (rval)) then
    var%rval = rval
    var%is_defined = .true.
    var%is_known = .true.
else
    var%is_known = .false.
end if
if (present (intrinsic)) var%is_intrinsic = intrinsic
if (present (user)) var%is_user_var = user
var%is_allocated = .true.
end subroutine var_entry_init_real

subroutine var_entry_init_cmplx (var, name, cval, intrinsic, user)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    complex(default), intent(in), optional :: cval
    logical, intent(in), optional :: intrinsic, user
    var%name = name
    var%type = V_CMPLX
    allocate (var%cval, var%is_known)
    if (present (cval)) then
        var%cval = cval
        var%is_defined = .true.
        var%is_known = .true.
    else
        var%is_known = .false.
    end if
    if (present (intrinsic)) var%is_intrinsic = intrinsic
    if (present (user)) var%is_user_var = user
    var%is_allocated = .true.
end subroutine var_entry_init_cmplx

subroutine var_entry_init_subevt (var, name, pval, intrinsic, user)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), optional :: pval
    logical, intent(in), optional :: intrinsic, user
    var%name = name
    var%type = V_SEV
    allocate (var%pval, var%is_known)
    if (present (pval)) then
        var%pval = pval
        var%is_defined = .true.
        var%is_known = .true.
    else
        var%is_known = .false.
    end if
    if (present (intrinsic)) var%is_intrinsic = intrinsic
    if (present (user)) var%is_user_var = user
    var%is_allocated = .true.
end subroutine var_entry_init_subevt

subroutine var_entry_init_pdg_array (var, name, aval, intrinsic, user)

```

```

type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name
type(pdg_array_t), intent(in), optional :: aval
logical, intent(in), optional :: intrinsic, user
var%name = name
var%type = V_PDG
allocate (var%aval, var%is_known)
if (present (aval)) then
    var%aval = aval
    var%is_defined = .true.
    var%is_known = .true.
else
    var%is_known = .false.
end if
if (present (intrinsic)) var%is_intrinsic = intrinsic
if (present (user)) var%is_user_var = user
var%is_allocated = .true.
end subroutine var_entry_init_pdg_array

subroutine var_entry_init_string (var, name, sval, intrinsic, user)
type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name
type(string_t), intent(in), optional :: sval
logical, intent(in), optional :: intrinsic, user
var%name = name
var%type = V_STR
allocate (var%sval, var%is_known)
if (present (sval)) then
    var%sval = sval
    var%is_defined = .true.
    var%is_known = .true.
else
    var%is_known = .false.
end if
if (present (intrinsic)) var%is_intrinsic = intrinsic
if (present (user)) var%is_user_var = user
var%is_allocated = .true.
end subroutine var_entry_init_string

```

Initialize an entry with a pointer to the value and, for numeric/logical values, a pointer to the `is_known` flag.

*(Variables: public)*+≡

```

public :: var_entry_init_log_ptr
public :: var_entry_init_int_ptr
public :: var_entry_init_real_ptr
public :: var_entry_init_cmplx_ptr
public :: var_entry_init_pdg_array_ptr
public :: var_entry_init_subevt_ptr
public :: var_entry_init_string_ptr

```

*(Variables: procedures)*+≡

```

subroutine var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name

```

```

logical, intent(in), target :: lval
logical, intent(in), target :: is_known
logical, intent(in), optional :: intrinsic
var%name = name
var%type = V_LOG
var%lval => lval
var%is_known => is_known
if (present (intrinsic)) var%is_intrinsic = intrinsic
var%is_defined = .true.
end subroutine var_entry_init_log_ptr

subroutine var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name
integer, intent(in), target :: ival
logical, intent(in), target :: is_known
logical, intent(in), optional :: intrinsic
var%name = name
var%type = V_INT
var%ival => ival
var%is_known => is_known
if (present (intrinsic)) var%is_intrinsic = intrinsic
var%is_defined = .true.
end subroutine var_entry_init_int_ptr

subroutine var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name
real(default), intent(in), target :: rval
logical, intent(in), target :: is_known
logical, intent(in), optional :: intrinsic
var%name = name
var%type = V_REAL
var%rval => rval
var%is_known => is_known
if (present (intrinsic)) var%is_intrinsic = intrinsic
var%is_defined = .true.
end subroutine var_entry_init_real_ptr

subroutine var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
type(var_entry_t), intent(out) :: var
type(string_t), intent(in) :: name
complex(default), intent(in), target :: cval
logical, intent(in), target :: is_known
logical, intent(in), optional :: intrinsic
var%name = name
var%type = V_CMPLX
var%cval => cval
var%is_known => is_known
if (present (intrinsic)) var%is_intrinsic = intrinsic
var%is_defined = .true.
end subroutine var_entry_init_cmplx_ptr

subroutine var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)

```

```

    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_PDG
    var%aval => aval
    var%is_known => is_known
    if (present (intrinsic)) var%is_intrinsic = intrinsic
    var%is_defined = .true.
end subroutine var_entry_init_pdg_array_ptr

subroutine var_entry_init_subevt_ptr (var, name, pval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), target :: pval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_SEV
    var%pval => pval
    var%is_known => is_known
    if (present (intrinsic)) var%is_intrinsic = intrinsic
    var%is_defined = .true.
end subroutine var_entry_init_subevt_ptr

subroutine var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_STR
    var%sval => sval
    var%is_known => is_known
    if (present (intrinsic)) var%is_intrinsic = intrinsic
    var%is_defined = .true.
end subroutine var_entry_init_string_ptr

```

Initialize an entry with an observable. The procedure pointer is not yet set.

*(Variables: procedures)*+≡

```

subroutine var_entry_init_obs (var, name, type, prt1, prt2)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    integer, intent(in) :: type
    type(prt_t), intent(in), target :: prt1
    type(prt_t), intent(in), optional, target :: prt2
    var%type = type
    var%name = name
    var%prt1 => prt1
    if (present (prt2)) var%prt2 => prt2

```



```

    var%is_intrinsic = .true.
    var%is_defined = .true.
end subroutine var_entry_init_obs

```

Mark an entry as undefined if it is a user-defined variable object, so force re-initialization.

```

<Variables: procedures>+≡
subroutine var_entry_undefine (var)
  type(var_entry_t), intent(inout) :: var
  var%is_defined = .not. var%is_user_var
  var%is_known = var%is_defined .and. var%is_known
end subroutine var_entry_undefine

```

Lock an entry: forbid resetting the entry after initialization.

```

<Variables: procedures>+≡
subroutine var_entry_lock (var, locked)
  type(var_entry_t), intent(inout) :: var
  logical, intent(in), optional :: locked
  if (present (locked)) then
    var%is_locked = locked
  else
    var%is_locked = .true.
  end if
end subroutine var_entry_lock

```

## Finalizer

```

<Variables: public>+≡
public :: var_entry_final

<Variables: procedures>+≡
subroutine var_entry_final (var)
  type(var_entry_t), intent(inout) :: var
  if (var%is_allocated) then
    select case (var%type)
      case (V_LOG); deallocate (var%lval)
      case (V_INT); deallocate (var%ival)
      case (V_REAL); deallocate (var%rval)
      case (V_CMPLX); deallocate (var%cval)
      case (V_SEV); deallocate (var%pval)
      case (V_PDG); deallocate (var%aval)
      case (V_STR); deallocate (var%sval)
    end select
    deallocate (var%is_known)
    var%is_allocated = .false.
    var%is_defined = .false.
  end if
end subroutine var_entry_final

```

## Output

*<Variables: public>+≡*

```
public :: var_entry_write
```

*<Variables: procedures>+≡*

```
recursive subroutine var_entry_write (var, unit, model_name, show_ptr, &
    intrinsic)
    type(var_entry_t), intent(in) :: var
    integer, intent(in), optional :: unit
    type(string_t), intent(in), optional :: model_name
    logical, intent(in), optional :: show_ptr
    logical, intent(in), optional :: intrinsic
    integer :: u
    u = output_unit (unit); if (u < 0) return
    if (present (intrinsic)) then
        if (var%is_intrinsic .neqv. intrinsic) return
    end if
    if (.not. var%is_defined) then
        write (u, "(A,1x)", advance="no") "[undefined]"
    end if
    if (.not. var%is_intrinsic) then
        write (u, "(A,1x)", advance="no") "[user variable]"
    end if
    if (associated (var%original)) then
        if (present (model_name)) then
            write (u, "(A,A)", advance="no") char(model_name), "."
        end if
    end if
    write (u, "(A)", advance="no") char (var%name)
    if (var%is_locked) write (u, "(A)", advance="no") "*"
    if (var%is_allocated) then
        write (u, "(A)", advance="no") " = "
    else if (var%type /= V_NONE) then
        write (u, "(A)", advance="no") " => "
    end if
    select case (var%type)
    case (V_NONE); write (u, *)
    case (V_LOG)
        if (var%is_known) then
            if (var%lval) then
                write (u, "(A)") "true"
            else
                write (u, "(A)") "false"
            end if
        else
            write (u, "(A)") "[unknown logical]"
        end if
    case (V_INT)
        if (var%is_known) then
            write (u, "(IO)") var%ival
        else
            write (u, "(A)") "[unknown integer]"
        end if
    case (V_REAL)
```

```

        if (var%is_known) then
            write (u, "(ES19.12)") var%rval
        else
            write (u, "(A)") "[unknown real]"
        end if
    case (V_CMPLX)
        if (var%is_known) then
            write (u, "('(',ES19.12,',',',',ES19.12,')')") var%cval
        else
            write (u, "(A)") "[unknown complex]"
        end if
    case (V_SEV)
        if (var%is_known) then
            call subevt_write (var%pval, unit, prefix="      ")
        else
            write (u, "(A)") "[unknown subevent]"
        end if
    case (V_PDG)
        if (var%is_known) then
            call pdg_array_write (var%aval, u); write (u, *)
        else
            write (u, "(A)") "[unknown PDG array]"
        end if
    case (V_STR)
        if (var%is_known) then
            write (u, "(A)") '""' // char (var%sval) // '""'
        else
            write (u, "(A)") "[unknown string]"
        end if
    case (V_OBS1_INT); write (u, *) "[int] = unary observable"
    case (V_OBS2_INT); write (u, *) "[int] = binary observable"
    case (V_OBS1_REAL); write (u, *) "[real] = unary observable"
    case (V_OBS2_REAL); write (u, *) "[real] = binary observable"
    case (V_UOBS1_INT); write (u, *) "[int] = unary user observable"
    case (V_UOBS2_INT); write (u, *) "[int] = binary user observable"
    case (V_UOBS1_REAL); write (u, *) "[real] = unary user observable"
    case (V_UOBS2_REAL); write (u, *) "[real] = binary user observable"
end select
if (present (show_ptr)) then
    if (show_ptr .and. var%is_copy .and. associated (var%original)) then
        write (u, "(' => ')", advance="no")
        call var_entry_write (var%original, unit)
    end if
end if
end subroutine var_entry_write

```

## Accessing contents

*<Variables: public>+≡*

```

    public :: var_entry_get_name
    public :: var_entry_get_type

```

*<Variables: procedures>+≡*

```

function var_entry_get_name (var) result (name)
    type(string_t) :: name
    type(var_entry_t), intent(in) :: var
    name = var%name
end function var_entry_get_name

function var_entry_get_type (var) result (type)
    integer :: type
    type(var_entry_t), intent(in) :: var
    type = var%type
end function var_entry_get_type

```

Return true if the variable is defined. This the case if it is allocated and known, or if it is a pointer.

```

<Variables: public>+≡
    public :: var_entry_is_defined

<Variables: procedures>+≡
    function var_entry_is_defined (var) result (defined)
        logical :: defined
        type(var_entry_t), intent(in) :: var
        defined = var%is_defined
    end function var_entry_is_defined

```

Return true if the variable is locked

```

<Variables: public>+≡
    public :: var_entry_is_locked

<Variables: procedures>+≡
    function var_entry_is_locked (var) result (locked)
        logical :: locked
        type(var_entry_t), intent(in) :: var
        locked = var%is_locked
    end function var_entry_is_locked

```

Return true if the variable is intrinsic

```

<Variables: public>+≡
    public :: var_entry_is_intrinsic

<Variables: procedures>+≡
    function var_entry_is_intrinsic (var) result (flag)
        logical :: flag
        type(var_entry_t), intent(in) :: var
        flag = var%is_intrinsic
    end function var_entry_is_intrinsic

```

Return true if the variable is a copy

```

<Variables: public>+≡
    public :: var_entry_is_copy

```

```

<Variables: procedures>+=
function var_entry_is_copy (var) result (flag)
    logical :: flag
    type(var_entry_t), intent(in) :: var
    flag = var%is_copy
end function var_entry_is_copy

```

Return components

```

<Variables: public>+=
public :: var_entry_is_known
public :: var_entry_get_lval
public :: var_entry_get_ival
public :: var_entry_get_rval
public :: var_entry_get_cval
public :: var_entry_get_aval
public :: var_entry_get_pval
public :: var_entry_get_sval

<Variables: procedures>+=
function var_entry_is_known (var) result (flag)
    logical :: flag
    type(var_entry_t), intent(in) :: var
    flag = var%is_known
end function var_entry_is_known

function var_entry_get_lval (var) result (lval)
    logical :: lval
    type(var_entry_t), intent(in) :: var
    lval = var%lval
end function var_entry_get_lval

function var_entry_get_ival (var) result (ival)
    integer :: ival
    type(var_entry_t), intent(in) :: var
    ival = var%ival
end function var_entry_get_ival

function var_entry_get_rval (var) result (rval)
    real(default) :: rval
    type(var_entry_t), intent(in) :: var
    rval = var%rval
end function var_entry_get_rval

function var_entry_get_cval (var) result (cval)
    complex(default) :: cval
    type(var_entry_t), intent(in) :: var
    cval = var%cval
end function var_entry_get_cval

function var_entry_get_aval (var) result (aval)
    type(pdg_array_t) :: aval
    type(var_entry_t), intent(in) :: var
    aval = var%aval
end function var_entry_get_aval

```

```

function var_entry_get_pval (var) result (pval)
    type(subevt_t) :: pval
    type(var_entry_t), intent(in) :: var
    pval = var%pval
end function var_entry_get_pval

function var_entry_get_sval (var) result (sval)
    type(string_t) :: sval
    type(var_entry_t), intent(in) :: var
    sval = var%sval
end function var_entry_get_sval

```

Return pointers to components

*(Variables: public)*+≡

```

public :: var_entry_get_known_ptr
public :: var_entry_get_lval_ptr
public :: var_entry_get_ival_ptr
public :: var_entry_get_rval_ptr
public :: var_entry_get_cval_ptr
public :: var_entry_get_aval_ptr
public :: var_entry_get_pval_ptr
public :: var_entry_get_sval_ptr

```

*(Variables: procedures)*+≡

```

function var_entry_get_known_ptr (var) result (ptr)
    logical, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%is_known
end function var_entry_get_known_ptr

function var_entry_get_lval_ptr (var) result (ptr)
    logical, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%lval
end function var_entry_get_lval_ptr

function var_entry_get_ival_ptr (var) result (ptr)
    integer, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%ival
end function var_entry_get_ival_ptr

function var_entry_get_rval_ptr (var) result (ptr)
    real(default), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%rval
end function var_entry_get_rval_ptr

function var_entry_get_cval_ptr (var) result (ptr)
    complex(default), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%cval
end function var_entry_get_cval_ptr

```

```

function var_entry_get_pval_ptr (var) result (ptr)
  type(subevt_t), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%pval
end function var_entry_get_pval_ptr

function var_entry_get_aval_ptr (var) result (ptr)
  type(pdg_array_t), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%aval
end function var_entry_get_aval_ptr

function var_entry_get_sval_ptr (var) result (ptr)
  type(string_t), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%sval
end function var_entry_get_sval_ptr

<Variables: public>+≡
  public :: var_entry_get_prt1_ptr
  public :: var_entry_get_prt2_ptr

<Variables: procedures>+≡
  function var_entry_get_prt1_ptr (var) result (ptr)
    type(prt_t), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%prt1
  end function var_entry_get_prt1_ptr

  function var_entry_get_prt2_ptr (var) result (ptr)
    type(prt_t), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%prt2
  end function var_entry_get_prt2_ptr

```

We would like to also use functions here (for consistency), but a nagfor bug temporarily forces use to use subroutines.

```

<Variables: public>+≡
  public :: var_entry_assign_obs1_int_ptr
  public :: var_entry_assign_obs1_real_ptr
  public :: var_entry_assign_obs2_int_ptr
  public :: var_entry_assign_obs2_real_ptr

<Variables: procedures>+≡
  subroutine var_entry_assign_obs1_int_ptr (ptr, var)
    procedure(obs_unary_int), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%obs1_int
  end subroutine var_entry_assign_obs1_int_ptr

  subroutine var_entry_assign_obs1_real_ptr (ptr, var)
    procedure(obs_unary_real), pointer :: ptr
    type(var_entry_t), intent(in), target :: var

```

```

ptr => var%obs1_real
end subroutine var_entry_assign_obs1_real_ptr

subroutine var_entry_assign_obs2_int_ptr (ptr, var)
  procedure(obs_binary_int), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%obs2_int
end subroutine var_entry_assign_obs2_int_ptr

subroutine var_entry_assign_obs2_real_ptr (ptr, var)
  procedure(obs_binary_real), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%obs2_real
end subroutine var_entry_assign_obs2_real_ptr

```

## 5.5.2 Setting values

Undefine the value.

*(Variables: procedures)*+≡

```

subroutine var_entry_clear_value (var)
  type(var_entry_t), intent(inout) :: var
  var%is_known = .false.
end subroutine var_entry_clear_value

```

*(Variables: public)*+≡

```

public :: var_entry_set_log
public :: var_entry_set_int
public :: var_entry_set_real
public :: var_entry_set_cmplx
public :: var_entry_set_pdg_array
public :: var_entry_set_subevt
public :: var_entry_set_string

```

*(Variables: procedures)*+≡

```

recursive subroutine var_entry_set_log &
  (var, lval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  logical, intent(in) :: lval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%lval = lval
  var%is_known = is_known
  var%is_defined = .true.
  if (associated (var%original)) then
    call var_entry_set_log (var%original, lval, is_known)
  end if
  if (present (verbose)) then
    if (verbose) then
      call var_entry_write (var, model_name=model_name)
    end if
  end if

```



```

        call var_entry_write (var, model_name=model_name, unit=u)
        if (u >= 0) flush (u)
    end if
end if
end subroutine var_entry_set_log

recursive subroutine var_entry_set_int &
    (var, ival, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    integer, intent(in) :: ival
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%ival = ival
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
        call var_entry_set_int (var%original, ival, is_known)
    end if
    if (present (verbose)) then
        if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
        end if
    end if
end subroutine var_entry_set_int

recursive subroutine var_entry_set_real &
    (var, rval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    real(default), intent(in) :: rval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%rval = rval
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
        call var_entry_set_real (var%original, rval, is_known)
    end if
    if (present (verbose)) then
        if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
        end if
    end if
end subroutine var_entry_set_real

```

```

recursive subroutine var_entry_set_cmplx &
    (var, cval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    complex(default), intent(in) :: cval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%cval = cval
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
        call var_entry_set_cmplx (var%original, cval, is_known)
    end if
    if (present (verbose)) then
        if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
        end if
    end if
end subroutine var_entry_set_cmplx

recursive subroutine var_entry_set_pdg_array &
    (var, aval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(pdg_array_t), intent(in) :: aval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%aval = aval
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
        call var_entry_set_pdg_array (var%original, aval, is_known)
    end if
    if (present (verbose)) then
        if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
        end if
    end if
end subroutine var_entry_set_pdg_array

recursive subroutine var_entry_set_subevt &
    (var, pval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(subevt_t), intent(in) :: pval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose

```

```

type(string_t), intent(in), optional :: model_name
integer :: u
u = logfile_unit ()
var%pval = pval
var%is_known = is_known
var%is_defined = .true.
if (associated (var%original)) then
    call var_entry_set_subevt (var%original, pval, is_known)
end if
if (present (verbose)) then
    if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        if (u >= 0) flush (u)
    end if
end if
end subroutine var_entry_set_subevt

recursive subroutine var_entry_set_string &
    (var, sval, is_known, verbose, model_name)
type(var_entry_t), intent(inout) :: var
type(string_t), intent(in) :: sval
logical, intent(in) :: is_known
logical, intent(in), optional :: verbose
type(string_t), intent(in), optional :: model_name
integer :: u
u = logfile_unit ()
var%sval = sval
var%is_known = is_known
var%is_defined = .true.
if (associated (var%original)) then
    call var_entry_set_string (var%original, sval, is_known)
end if
if (present (verbose)) then
    if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        if (u >= 0) flush (u)
    end if
end if
end subroutine var_entry_set_string

```

### 5.5.3 Copies and pointer variables

Initialize an entry with a copy of an existing variable entry. The copy is physically allocated with the same type as the original.

*(Variables: procedures)+≡*

```

subroutine var_entry_init_copy (var, original, user)
type(var_entry_t), intent(out) :: var
type(var_entry_t), intent(in), target :: original
logical, intent(in), optional :: user
type(string_t) :: name

```

```

logical :: intrinsic
name = var_entry_get_name (original)
intrinsic = original%is_intrinsic
select case (original%type)
case (V_LOG)
    call var_entry_init_log (var, name, intrinsic=intrinsic, user=user)
case (V_INT)
    call var_entry_init_int (var, name, intrinsic=intrinsic, user=user)
case (V_REAL)
    call var_entry_init_real (var, name, intrinsic=intrinsic, user=user)
case (V_CMPLX)
    call var_entry_init_cmplx (var, name, intrinsic=intrinsic, user=user)
case (V_SEV)
    call var_entry_init_subevt (var, name, intrinsic=intrinsic, user=user)
case (V_PDG)
    call var_entry_init_pdg_array (var, name, intrinsic=intrinsic, user=user)
case (V_STR)
    call var_entry_init_string (var, name, intrinsic=intrinsic, user=user)
end select
var%is_copy = .true.
end subroutine var_entry_init_copy

```

Clear the pointer to the original.

```

<Variables: procedures> +=
subroutine var_entry_clear_original_pointer (var)
    type(var_entry_t), intent(inout) :: var
    var%original => null ()
end subroutine var_entry_clear_original_pointer

```

Set the pointer to the original. For a free parameter, the variable holds both the value and the pointer. For a derived parameter, we associate the pointer directly. Derived parameters thus need not be synchronized explicitly.

Update: this does not work. If locked parameters are accessed in expressions and the model is non-default, the pointers in the expression may be undefined at compile time. Reassigning the variables at runtime does not help, since the pointers in the expression are dereferenced before assignment. Hence, no special treatment for derived parameters.

```

<Variables: procedures> +=
subroutine var_entry_set_original_pointer (var, original)
    type(var_entry_t), intent(inout) :: var
    type(var_entry_t), intent(in), target :: original
    type(string_t) :: name
    type(var_entry_t), pointer :: next
    if (var_entry_is_locked (original)) then
!       next => var%next
!       name = var_entry_get_name (original)
!       select case (original%type)
!       case (V_LOG); call var_entry_init_log_ptr (var, name, &
!           original%lval, original%is_known)
!       case (V_INT); call var_entry_init_int_ptr (var, name, &
!           original%ival, original%is_known)
!       case (V_REAL); call var_entry_init_real_ptr (var, name, &

```

```

!         original%rval, original%is_known)
!     case (V_CMPLX); call var_entry_init_cmplx_ptr (var, name, &
!         original%cval, original%is_known)
!     case (V_SEV);  call var_entry_init_subevt_ptr (var, name, &
!         original%pval, original%is_known)
!     case (V_PDG);  call var_entry_init_pdg_array_ptr (var, name, &
!         original%aval, original%is_known)
!     case (V_STR);  call var_entry_init_string_ptr (var, name, &
!         original%sval, original%is_known)
!     end select
!     var%next => next
!     call var_entry_lock (var)
!     else
!         var%original => original
!     end if
!     var%original => original
end subroutine var_entry_set_original_pointer

```

Synchronize a variable with its original: if a pointer exists, set the value to be equal to value pointed to.

*(Variables: procedures)+≡*

```

subroutine var_entry_synchronize (var)
  type(var_entry_t), intent(inout) :: var
  if (associated (var%original)) then
    var%is_defined = var%original%is_defined
    var%is_known = var%original%is_known
    if (var%original%is_known) then
      select case (var%type)
        case (V_LOG);  var%lval = var%original%lval
        case (V_INT);  var%ival = var%original%ival
        case (V_REAL); var%rval = var%original%rval
        case (V_CMPLX); var%cval = var%original%cval
        case (V_SEV);  var%pval = var%original%pval
        case (V_PDG);  var%aval = var%original%aval
        case (V_STR);  var%sval = var%original%sval
      end select
    end if
  end if
end subroutine var_entry_synchronize

```

Restore the previous value of the original, using the value stored in the variable. This is a side-effect operation.

*(Variables: procedures)+≡*

```

subroutine var_entry_restore (var)
  type(var_entry_t), intent(inout) :: var
  !!! ifort 11.1 rev5 chokes over the intent(in)
  !!! type(var_entry_t), intent(in) :: var
  if (associated (var%original)) then
    if (var%is_known) then
      select case (var%type)
        case (V_LOG);  var%original%lval = var%lval
        case (V_INT);  var%original%ival = var%ival
        case (V_REAL); var%original%rval = var%rval

```

```

        case (V_CMPLX); var%original%cval = var%cval
        case (V_SEV);  var%original%pval = var%pval
        case (V_PDG);  var%original%aval = var%aval
        case (V_STR);  var%original%sval = var%sval
      end select
    end if
  end if
end subroutine var_entry_restore

```

### 5.5.4 Variable lists

#### The type

Variable lists can be linked together. No initializer needed. They are deleted separately.

```

<Variables: public>+≡
  public :: var_list_t

<Variables: types>+≡
  type :: var_list_t
    private
    type(var_entry_t), pointer :: first => null ()
    type(var_entry_t), pointer :: last => null ()
    type(var_list_t), pointer :: next => null ()
  end type var_list_t

```

#### Constructors

```

<Variables: public>+≡
  public :: var_list_link

<Variables: procedures>+≡
  subroutine var_list_link (var_list, next)
    type(var_list_t), intent(inout) :: var_list
    type(var_list_t), intent(in), target :: next
    var_list%next => next
  end subroutine var_list_link

```

Append a new entry to an existing list.

```

<Variables: procedures>+≡
  subroutine var_list_append (var_list, var, verbose)
    type(var_list_t), intent(inout) :: var_list
    type(var_entry_t), intent(in), target :: var
    logical, intent(in), optional :: verbose
    if (associated (var_list%last)) then
      var_list%last%next => var
    else
      var_list%first => var
    end if
    var_list%last => var
    if (present (verbose)) then
      if (verbose) call var_entry_write (var)
    end if
  end subroutine var_list_append

```

```

        end if
    end subroutine var_list_append

    {Variables: public}+=
    public :: var_list_append_log
    public :: var_list_append_int
    public :: var_list_append_real
    public :: var_list_append_cmplx
    public :: var_list_append_subevt
    public :: var_list_append_pdg_array
    public :: var_list_append_string

    {Variables: interfaces}+=
    interface var_list_append_log
        module procedure var_list_append_log_s
        module procedure var_list_append_log_c
    end interface
    interface var_list_append_int
        module procedure var_list_append_int_s
        module procedure var_list_append_int_c
    end interface
    interface var_list_append_real
        module procedure var_list_append_real_s
        module procedure var_list_append_real_c
    end interface
    interface var_list_append_cmplx
        module procedure var_list_append_cmplx_s
        module procedure var_list_append_cmplx_c
    end interface
    interface var_list_append_subevt
        module procedure var_list_append_subevt_s
        module procedure var_list_append_subevt_c
    end interface
    interface var_list_append_pdg_array
        module procedure var_list_append_pdg_array_s
        module procedure var_list_append_pdg_array_c
    end interface
    interface var_list_append_string
        module procedure var_list_append_string_s
        module procedure var_list_append_string_c
    end interface

    {Variables: procedures}+=
    subroutine var_list_append_log_s &
        (var_list, name, lval, locked, verbose, intrinsic, user)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        logical, intent(in), optional :: lval
        logical, intent(in), optional :: locked, verbose, intrinsic, user
        type(var_entry_t), pointer :: var
        allocate (var)
        call var_entry_init_log (var, name, lval, intrinsic, user)
        if (present (locked)) call var_entry_lock (var, locked)
        call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_log_s

```

```

subroutine var_list_append_int_s &
    (var_list, name, ival, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    integer, intent(in), optional :: ival
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_int (var, name, ival, intrinsic, user)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_int_s

subroutine var_list_append_real_s &
    (var_list, name, rval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    real(default), intent(in), optional :: rval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_real (var, name, rval, intrinsic, user)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_real_s

subroutine var_list_append_cmplx_s &
    (var_list, name, cval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    complex(default), intent(in), optional :: cval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_cmplx (var, name, cval, intrinsic, user)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_cmplx_s

subroutine var_list_append_subevt_s &
    (var_list, name, pval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), optional :: pval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_subevt (var, name, pval, intrinsic, user)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_subevt_s

subroutine var_list_append_pdg_array_s &

```



```

        (var_list, name, aval, locked, verbose, intrinsic, user)
type(var_list_t), intent(inout) :: var_list
type(string_t), intent(in) :: name
type(pdg_array_t), intent(in), optional :: aval
logical, intent(in), optional :: locked, verbose, intrinsic, user
type(var_entry_t), pointer :: var
allocate (var)
call var_entry_init_pdg_array (var, name, aval, intrinsic, user)
if (present (locked)) call var_entry_lock (var, locked)
call var_list_append (var_list, var, verbose)
end subroutine var_list_append_pdg_array_s

subroutine var_list_append_string_s &
    (var_list, name, sval, locked, verbose, intrinsic, user)
type(var_list_t), intent(inout) :: var_list
type(string_t), intent(in) :: name
type(string_t), intent(in), optional :: sval
logical, intent(in), optional :: locked, verbose, intrinsic, user
type(var_entry_t), pointer :: var
allocate (var)
call var_entry_init_string (var, name, sval, intrinsic, user)
if (present (locked)) call var_entry_lock (var, locked)
call var_list_append (var_list, var, verbose)
end subroutine var_list_append_string_s

subroutine var_list_append_log_c &
    (var_list, name, lval, locked, verbose, intrinsic, user)
type(var_list_t), intent(inout) :: var_list
character(*), intent(in) :: name
logical, intent(in), optional :: lval
logical, intent(in), optional :: locked, verbose, intrinsic, user
call var_list_append_log_s &
    (var_list, var_str (name), lval, locked, verbose, intrinsic, user)
end subroutine var_list_append_log_c

subroutine var_list_append_int_c &
    (var_list, name, ival, locked, verbose, intrinsic, user)
type(var_list_t), intent(inout) :: var_list
character(*), intent(in) :: name
integer, intent(in), optional :: ival
logical, intent(in), optional :: locked, verbose, intrinsic, user
call var_list_append_int_s &
    (var_list, var_str (name), ival, locked, verbose, intrinsic, user)
end subroutine var_list_append_int_c

subroutine var_list_append_real_c &
    (var_list, name, rval, locked, verbose, intrinsic, user)
type(var_list_t), intent(inout) :: var_list
character(*), intent(in) :: name
real(default), intent(in), optional :: rval
logical, intent(in), optional :: locked, verbose, intrinsic, user
call var_list_append_real_s &
    (var_list, var_str (name), rval, locked, verbose, intrinsic, user)
end subroutine var_list_append_real_c

```

```

subroutine var_list_append_cmplx_c &
    (var_list, name, cval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    character(*), intent(in) :: name
    complex(default), intent(in), optional :: cval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    call var_list_append_cmplx_s &
        (var_list, var_str (name), cval, locked, verbose, intrinsic, user)
end subroutine var_list_append_cmplx_c

subroutine var_list_append_subevt_c &
    (var_list, name, pval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    character(*), intent(in) :: name
    type(subevt_t), intent(in), optional :: pval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    call var_list_append_subevt_s &
        (var_list, var_str (name), pval, locked, verbose, intrinsic, user)
end subroutine var_list_append_subevt_c

subroutine var_list_append_pdg_array_c &
    (var_list, name, aval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    character(*), intent(in) :: name
    type(pdg_array_t), intent(in), optional :: aval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    call var_list_append_pdg_array_s &
        (var_list, var_str (name), aval, locked, verbose, intrinsic, user)
end subroutine var_list_append_pdg_array_c

subroutine var_list_append_string_c &
    (var_list, name, sval, locked, verbose, intrinsic, user)
    type(var_list_t), intent(inout) :: var_list
    character(*), intent(in) :: name
    character(*), intent(in), optional :: sval
    logical, intent(in), optional :: locked, verbose, intrinsic, user
    if (present (sval)) then
        call var_list_append_string_s &
            (var_list, var_str (name), var_str (sval), &
                locked, verbose, intrinsic, user)
    else
        call var_list_append_string_s &
            (var_list, var_str (name), &
                locked=locked, verbose=verbose, intrinsic=intrinsic, user=user)
    end if
end subroutine var_list_append_string_c

```

*<Variables: public>+≡*

```

public :: var_list_append_log_ptr
public :: var_list_append_int_ptr
public :: var_list_append_real_ptr
public :: var_list_append_cmplx_ptr
public :: var_list_append_pdg_array_ptr

```

```

public :: var_list_append_subevt_ptr
public :: var_list_append_string_ptr

{Variables: procedures}+=
subroutine var_list_append_log_ptr &
    (var_list, name, lval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    logical, intent(in), target :: lval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_log_ptr

subroutine var_list_append_int_ptr &
    (var_list, name, ival, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    integer, intent(in), target :: ival
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_int_ptr

subroutine var_list_append_real_ptr &
    (var_list, name, rval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    real(default), intent(in), target :: rval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_real_ptr

subroutine var_list_append_cmplx_ptr &
    (var_list, name, cval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    complex(default), intent(in), target :: cval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)

```

```

    call var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_cmplx_ptr

subroutine var_list_append_pdg_array_ptr &
    (var_list, name, aval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_pdg_array_ptr

subroutine var_list_append_subevt_ptr &
    (var_list, name, pval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), target :: pval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_subevt_ptr (var, name, pval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_subevt_ptr

subroutine var_list_append_string_ptr &
    (var_list, name, sval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
    if (present (locked)) call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_string_ptr

```

## Finalizer

```

<Variables: public>+≡
    public :: var_list_final

```

Finalize, delete the list entry by entry.

```

<Variables: procedures>+≡
subroutine var_list_final (var_list)
  type(var_list_t), intent(inout) :: var_list
  type(var_entry_t), pointer :: var
  var_list%last => null ()
  do while (associated (var_list%first))
    var => var_list%first
    var_list%first => var%next
    call var_entry_final (var)
    deallocate (var)
  end do
end subroutine var_list_final

```

## Output

Optionally, show only variables of a certain type.

```

<Variables: public>+≡
public :: var_list_write

<Variables: procedures>+≡
recursive subroutine var_list_write &
  (var_list, unit, follow_link, only_type, prefix, model_name, show_ptr, &
   intrinsic)
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: follow_link
  integer, intent(in), optional :: only_type
  character(*), intent(in), optional :: prefix
  type(string_t), intent(in), optional :: model_name
  logical, intent(in), optional :: show_ptr
  logical, intent(in), optional :: intrinsic
  type(var_entry_t), pointer :: var
  integer :: u, length
  logical :: write_this, write_next
  u = output_unit (unit); if (u < 0) return
  if (present (prefix)) length = len (prefix)
  var => var_list%first
  if (associated (var)) then
    do while (associated (var))
      if (present (only_type)) then
        write_this = only_type == var%type
      else
        write_this = .true.
      end if
      if (write_this .and. present (prefix)) then
        if (prefix /= extract (var%name, 1, length)) &
          write_this = .false.
      end if
      if (write_this) then
        call var_entry_write &
          (var, unit, model_name = model_name, show_ptr = show_ptr, &
           intrinsic=intrinsic)
      end if
    end do
  end if
end subroutine var_list_write

```

```

        var => var%next
    end do
end if
write_next = associated (var_list%next)
if (present (follow_link)) &
    write_next = write_next .and. follow_link
if (write_next) then
    call var_list_write (var_list%next, &
        unit, follow_link, only_type, prefix, model_name, show_ptr, &
        intrinsic)
end if
end subroutine var_list_write

```

Write only a certain variable.

```

<Variables: public>+≡
    public :: var_list_write_var

<Variables: procedures>+≡
    recursive subroutine var_list_write_var &
        (var_list, name, unit, type, follow_link, model_name, show_ptr)
    type(var_list_t), intent(in), target :: var_list
    type(string_t), intent(in) :: name
    integer, intent(in), optional :: unit
    integer, intent(in), optional :: type
    logical, intent(in), optional :: follow_link
    type(string_t), intent(in), optional :: model_name
    logical, intent(in), optional :: show_ptr
    type(var_entry_t), pointer :: var
    integer :: u
    u = output_unit (unit); if (u < 0) return
    var => var_list_get_var_ptr &
        (var_list, name, type, follow_link=follow_link, defined=.true.)
    if (associated (var)) then
        call var_entry_write &
            (var, unit, model_name = model_name, show_ptr = show_ptr)
    else
        write (u, "(A)") char (name) // " = [undefined]"
    end if
end subroutine var_list_write_var

```

### 5.5.5 Tools

Return a pointer to the variable list linked to by the current one.

```

<Variables: public>+≡
    public :: var_list_get_next_ptr

<Variables: procedures>+≡
    function var_list_get_next_ptr (var_list) result (next_ptr)
    type(var_list_t), pointer :: next_ptr
    type(var_list_t), intent(in) :: var_list
    next_ptr => var_list%next
end function var_list_get_next_ptr

```

Return a pointer to the variable with the requested name. If no such name exists, return a null pointer. In that case, try the next list if present, unless follow\_link is unset. If defined is set, ignore entries that exist but are undefined.

```

<Variables: public>+=
    public :: var_list_get_var_ptr

<Variables: procedures>+=
    recursive function var_list_get_var_ptr &
        (var_list, name, type, follow_link, defined) result (var)
        type(var_entry_t), pointer :: var
        type(var_list_t), intent(in), target :: var_list
        type(string_t), intent(in) :: name
        integer, intent(in), optional :: type
        logical, intent(in), optional :: follow_link, defined
        logical :: ignore_undef, search_next
        ignore_undef = .true.; if (present (defined)) ignore_undef = .not. defined
        var => var_list%first
        if (present (type)) then
            do while (associated (var))
                if (var%type == type) then
                    if (var%name == name) then
                        if (ignore_undef .or. var%is_defined) return
                    end if
                end if
                var => var%next
            end do
        else
            do while (associated (var))
                if (var%name == name) then
                    if (ignore_undef .or. var%is_defined) return
                end if
                var => var%next
            end do
        end if
        search_next = associated (var_list%next)
        if (present (follow_link)) &
            search_next = search_next .and. follow_link
        if (search_next) &
            var => var_list_get_var_ptr &
                (var_list%next, name, type, defined=defined)
    end function var_list_get_var_ptr

```

Return the variable type

```

<Variables: public>+=
    public :: var_list_get_type

<Variables: procedures>+=
    function var_list_get_type (var_list, name, follow_link) result (type)
        integer :: type
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var

```

```

var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
if (associated (var)) then
    type = var%type
else
    type = V_NONE
end if
end function var_list_get_type

```

Return true if the variable exists.

*<Variables: public>+≡*

```
public :: var_list_exists
```

*<Variables: procedures>+≡*

```

function var_list_exists (var_list, name, follow_link) result (flag)
    logical :: flag
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
    flag = associated (var)
end function var_list_exists

```

Return true if the variable is declared as intrinsic.

*<Variables: public>+≡*

```
public :: var_list_is_intrinsic
```

*<Variables: procedures>+≡*

```

function var_list_is_intrinsic (var_list, name, follow_link) result (flag)
    logical :: flag
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
    if (associated (var)) then
        flag = var%is_intrinsic
    else
        flag = .false.
    end if
end function var_list_is_intrinsic

```

Return true if the value is known.

*<Variables: public>+≡*

```
public :: var_list_is_known
```

*<Variables: interfaces>+≡*

```

interface var_list_is_known
    module procedure var_list_is_known_s
    module procedure var_list_is_known_c
end interface

```



```

{Variables: procedures}+=
function var_list_is_known_s (var_list, name, follow_link) result (flag)
    logical :: flag
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
    if (associated (var)) then
        flag = var%is_known
    else
        flag = .false.
    end if
end function var_list_is_known_s

function var_list_is_known_c (var_list, name, follow_link) result (flag)
    logical :: flag
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    flag = var_list_is_known_s (var_list, var_str (name), follow_link)
end function var_list_is_known_c

```

Return true if the value is locked.

```

{Variables: public}+=
public :: var_list_is_locked

{Variables: procedures}+=
function var_list_is_locked (var_list, name, follow_link) result (flag)
    logical :: flag
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
    if (associated (var)) then
        flag = var_entry_is_locked (var)
    else
        flag = .false.
    end if
end function var_list_is_locked

```

Return the value, assuming that the type is correct. We consider only variable entries that have been defined.

For convenience, allow both variable and fixed-length (literal) strings.

```

{Variables: public}+=
public :: var_list_get_lval
public :: var_list_get_ival
public :: var_list_get_rval
public :: var_list_get_cval
public :: var_list_get_pval
public :: var_list_get_aval
public :: var_list_get_sval

```

*<Variables: interfaces>+≡*

```

interface var_list_get_lval
  module procedure var_list_get_lval_s
  module procedure var_list_get_lval_c
end interface
interface var_list_get_ival
  module procedure var_list_get_ival_s
  module procedure var_list_get_ival_c
end interface
interface var_list_get_rval
  module procedure var_list_get_rval_s
  module procedure var_list_get_rval_c
end interface
interface var_list_get_cval
  module procedure var_list_get_cval_s
  module procedure var_list_get_cval_c
end interface
interface var_list_get_pval
  module procedure var_list_get_pval_s
  module procedure var_list_get_pval_c
end interface
interface var_list_get_aval
  module procedure var_list_get_aval_s
  module procedure var_list_get_aval_c
end interface
interface var_list_get_sval
  module procedure var_list_get_sval_s
  module procedure var_list_get_sval_c
end interface

```

*<Variables: procedures>+≡*

```

function var_list_get_lval_s (var_list, name, follow_link) result (lval)
  logical :: lval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr &
    (var_list, name, V_LOG, follow_link, defined=.true.)
  if (associated (var)) then
    if (var_has_value (var)) then
      lval = var%lval
    else
      lval = .false.
    end if
  else
    lval = .false.
  end if
end function var_list_get_lval_s

function var_list_get_ival_s (var_list, name, follow_link) result (ival)
  integer :: ival
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list

```

```

logical, intent(in), optional :: follow_link
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr &
    (var_list, name, V_INT, follow_link, defined=.true.)
if (associated (var)) then
    if (var_has_value (var)) then
        ival = var%ival
    else
        ival = 0
    end if
else
    ival = 0
end if
end function var_list_get_ival_s

function var_list_get_rval_s (var_list, name, follow_link) result (rval)
    real(default) :: rval
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr &
        (var_list, name, V_REAL, follow_link, defined=.true.)
    if (associated (var)) then
        if (var_has_value (var)) then
            rval = var%rval
        else
            rval = 0
        end if
    else
        rval = 0
    end if
end function var_list_get_rval_s

function var_list_get_cval_s (var_list, name, follow_link) result (cval)
    complex(default) :: cval
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr &
        (var_list, name, V_CMPLX, follow_link, defined=.true.)
    if (associated (var)) then
        if (var_has_value (var)) then
            cval = var%cval
        else
            cval = 0
        end if
    else
        cval = 0
    end if
end function var_list_get_cval_s

function var_list_get_aval_s (var_list, name, follow_link) result (aval)

```

```

type(pdg_array_t) :: aval
type(string_t), intent(in) :: name
type(var_list_t), intent(in), target :: var_list
logical, intent(in), optional :: follow_link
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr &
    (var_list, name, V_PDG, follow_link, defined=.true.)
if (associated (var)) then
    if (var_has_value (var)) then
        aval = var%aval
    end if
end if
end function var_list_get_aval_s

function var_list_get_pval_s (var_list, name, follow_link) result (pval)
    type(subvt_t) :: pval
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr &
        (var_list, name, V_SEV, follow_link, defined=.true.)
    if (associated (var)) then
        if (var_has_value (var)) then
            pval = var%pval
        end if
    end if
end function var_list_get_pval_s

function var_list_get_sval_s (var_list, name, follow_link) result (sval)
    type(string_t) :: sval
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr &
        (var_list, name, V_STR, follow_link, defined=.true.)
    if (associated (var)) then
        if (var_has_value (var)) then
            sval = var%sval
        else
            sval = ""
        end if
    else
        sval = ""
    end if
end function var_list_get_sval_s

function var_list_get_lval_c (var_list, name, follow_link) result (lval)
    logical :: lval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    lval = var_list_get_lval_s (var_list, var_str (name), follow_link)

```

```

end function var_list_get_lval_c

function var_list_get_ival_c (var_list, name, follow_link) result (ival)
    integer :: ival
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    ival = var_list_get_ival_s (var_list, var_str (name), follow_link)
end function var_list_get_ival_c

function var_list_get_rval_c (var_list, name, follow_link) result (rval)
    real(default) :: rval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    rval = var_list_get_rval_s (var_list, var_str (name), follow_link)
end function var_list_get_rval_c

function var_list_get_cval_c (var_list, name, follow_link) result (cval)
    complex(default) :: cval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    cval = var_list_get_cval_s (var_list, var_str (name), follow_link)
end function var_list_get_cval_c

function var_list_get_aval_c (var_list, name, follow_link) result (aval)
    type(pdg_array_t) :: aval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    aval = var_list_get_aval_s (var_list, var_str (name), follow_link)
end function var_list_get_aval_c

function var_list_get_pval_c (var_list, name, follow_link) result (pval)
    type(subevt_t) :: pval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    pval = var_list_get_pval_s (var_list, var_str (name), follow_link)
end function var_list_get_pval_c

function var_list_get_sval_c (var_list, name, follow_link) result (sval)
    type(string_t) :: sval
    character(*), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    logical, intent(in), optional :: follow_link
    sval = var_list_get_sval_s (var_list, var_str (name), follow_link)
end function var_list_get_sval_c

```

Check for a valid value, given a pointer. Issue error messages if invalid.

*<Variables: procedures>+≡*

```
function var_has_value (var) result (valid)
```

```

logical :: valid
type(var_entry_t), pointer :: var
if (associated (var)) then
  if (var%is_known) then
    valid = .true.
  else
    call msg_error ("The value of variable '" // char (var%name) &
      // "' is unknown but must be known at this point.")
    valid = .false.
  end if
else
  call msg_error ("Variable '" // char (var%name) &
    // "' is undefined but must have a known value at this point.")
  valid = .false.
end if
end function var_has_value

```

### 5.5.6 Process-specific variables

We allow the user to set a numeric process ID for each declared process.

```

<Variables: public>+≡
  public :: var_list_init_num_id

<Variables: procedures>+≡
  subroutine var_list_init_num_id (var_list, proc_id, num_id)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: proc_id
    integer, intent(in), optional :: num_id
    call var_list_set_procvar_int (var_list, proc_id, &
      var_str ("num_id"), num_id)
  end subroutine var_list_init_num_id

```

Integration results are stored in special variables. They are initialized by this subroutine. The values may or may not already be known.

```

<Variables: public>+≡
  public :: var_list_init_process_results

<Variables: procedures>+≡
  subroutine var_list_init_process_results (var_list, proc_id, &
    n_calls, integral, error, accuracy, chi2, efficiency)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: proc_id
    integer, intent(in), optional :: n_calls
    real(default), intent(in), optional :: integral, error, accuracy
    real(default), intent(in), optional :: chi2, efficiency
    call var_list_set_procvar_int (var_list, proc_id, &
      var_str ("n_calls"), n_calls)
    call var_list_set_procvar_real (var_list, proc_id, &
      var_str ("integral"), integral)
    call var_list_set_procvar_real (var_list, proc_id, &
      var_str ("error"), error)
    call var_list_set_procvar_real (var_list, proc_id, &
      var_str ("accuracy"), accuracy)

```

```

    call var_list_set_procvar_real (var_list, proc_id, &
        var_str ("chi2"), chi2)
    call var_list_set_procvar_real (var_list, proc_id, &
        var_str ("efficiency"), efficiency)
end subroutine var_list_init_process_results

```

*(Variables: procedures)+≡*

```

subroutine var_list_set_procvar_int (var_list, proc_id, name, ival)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: proc_id
    type(string_t), intent(in) :: name
    integer, intent(in), optional :: ival
    type(string_t) :: var_name
    type(var_entry_t), pointer :: var
    var_name = name // "(" // proc_id // ")"
    var => var_list_get_var_ptr (var_list, var_name)
    if (.not. associated (var)) then
        call var_list_append_int (var_list, var_name, ival, intrinsic=.true.)
    else if (present (ival)) then
        call var_list_set_int (var_list, var_name, ival, is_known=.true.)
    end if
end subroutine var_list_set_procvar_int

subroutine var_list_set_procvar_real (var_list, proc_id, name, rval)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: proc_id
    type(string_t), intent(in) :: name
    real(default), intent(in), optional :: rval
    type(string_t) :: var_name
    type(var_entry_t), pointer :: var
    var_name = name // "(" // proc_id // ")"
    var => var_list_get_var_ptr (var_list, var_name)
    if (.not. associated (var)) then
        call var_list_append_real (var_list, var_name, rval, intrinsic=.true.)
    else if (present (rval)) then
        call var_list_set_real (var_list, var_name, rval, is_known=.true.)
    end if
end subroutine var_list_set_procvar_real

```

### 5.5.7 Observable initialization

Observables are formally treated as variables, which however are evaluated each time the observable is used. The arguments (pointers) to evaluate and the function are part of the variable-list entry.

The procedure pointer should be set by this subroutine. This, however, triggers a bug in nagfor 5.2(649). As a workaround, we return the variable pointer, so the pointer can be set directly.

*(Variables: procedures)+≡*

```

subroutine var_list_set_obs (var_list, name, type, var, prt1, prt2)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name

```

```

integer, intent(in) :: type
type(var_entry_t), pointer :: var
type(prt_t), intent(in), target :: prt1
type(prt_t), intent(in), optional, target :: prt2
allocate (var)
call var_entry_init_obs (var, name, type, prt1, prt2)
call var_list_append (var_list, var)
end subroutine var_list_set_obs

```

Unary and binary observables are different. Most unary observables can be equally well evaluated for particle pairs. Binary observables cannot be evaluated for single particles.

*(Variables: public)+≡*

```

public :: var_list_set_observables_unary
public :: var_list_set_observables_binary

```

*(Variables: procedures)+≡*

```

subroutine var_list_set_observables_unary (var_list, prt1)
  type(var_list_t), intent(inout) :: var_list
  type(prt_t), intent(in), target :: prt1
  type(var_entry_t), pointer :: var
  call var_list_set_obs &
    (var_list, var_str ("PDG"), V_OBS1_INT, var, prt1)
  var% obs1_int => obs_pdg1
  call var_list_set_obs &
    (var_list, var_str ("Hel"), V_OBS1_INT, var, prt1)
  var% obs1_int => obs_helicity1
  call var_list_set_obs &
    (var_list, var_str ("M"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_signed_mass1
  call var_list_set_obs &
    (var_list, var_str ("M2"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_mass_squared1
  call var_list_set_obs &
    (var_list, var_str ("E"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_energy1
  call var_list_set_obs &
    (var_list, var_str ("Px"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_px1
  call var_list_set_obs &
    (var_list, var_str ("Py"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_py1
  call var_list_set_obs &
    (var_list, var_str ("Pz"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_pz1
  call var_list_set_obs &
    (var_list, var_str ("P"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_p1
  call var_list_set_obs &
    (var_list, var_str ("P1"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_pl1
  call var_list_set_obs &
    (var_list, var_str ("Pt"), V_OBS1_REAL, var, prt1)
  var% obs1_real => obs_pt1

```



```

call var_list_set_obs &
  (var_list, var_str ("Theta"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_theta1
call var_list_set_obs &
  (var_list, var_str ("Phi"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_phi1
call var_list_set_obs &
  (var_list, var_str ("Rap"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_rap1
call var_list_set_obs &
  (var_list, var_str ("Eta"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_eta1
call var_list_set_obs &
  (var_list, var_str ("Theta_RF"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_theta_rf1
call var_list_set_obs &
  (var_list, var_str ("Dist"), V_OBS1_REAL, var, prt1)
var% obs1_real => obs_dist1
call var_list_set_obs &
  (var_list, var_str ("_User_obs_real"), V_UOBS1_REAL, var, prt1)
call var_list_set_obs &
  (var_list, var_str ("_User_obs_int"), V_UOBS1_INT, var, prt1)
end subroutine var_list_set_observables_unary

subroutine var_list_set_observables_binary (var_list, prt1, prt2)
  type(var_list_t), intent(inout) :: var_list
  type(prt_t), intent(in), target :: prt1
  type(prt_t), intent(in), optional, target :: prt2
  type(var_entry_t), pointer :: var
  call var_list_set_obs &
    (var_list, var_str ("PDG"), V_OBS2_INT, var, prt1, prt2)
  var% obs2_int => obs_pdg2
  call var_list_set_obs &
    (var_list, var_str ("Hel"), V_OBS2_INT, var, prt1, prt2)
  var% obs2_int => obs_helicity2
  call var_list_set_obs &
    (var_list, var_str ("M"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_signed_mass2
  call var_list_set_obs &
    (var_list, var_str ("M2"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_mass_squared2
  call var_list_set_obs &
    (var_list, var_str ("E"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_energy2
  call var_list_set_obs &
    (var_list, var_str ("Px"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_px2
  call var_list_set_obs &
    (var_list, var_str ("Py"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_py2
  call var_list_set_obs &
    (var_list, var_str ("Pz"), V_OBS2_REAL, var, prt1, prt2)
  var% obs2_real => obs_pz2
  call var_list_set_obs &

```

```

        (var_list, var_str ("P"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_p2
call var_list_set_obs &
    (var_list, var_str ("P1"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_p12
call var_list_set_obs &
    (var_list, var_str ("Pt"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_pt2
call var_list_set_obs &
    (var_list, var_str ("Theta"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_theta2
call var_list_set_obs &
    (var_list, var_str ("Phi"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_phi2
call var_list_set_obs &
    (var_list, var_str ("Rap"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_rap2
call var_list_set_obs &
    (var_list, var_str ("Eta"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_eta2
call var_list_set_obs &
    (var_list, var_str ("Theta_RF"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_theta_rf2
call var_list_set_obs &
    (var_list, var_str ("Dist"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_dist2
call var_list_set_obs &
    (var_list, var_str ("kT"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_ktmeasure
call var_list_set_obs &
    (var_list, var_str ("_User_obs_real"), V_UOBS2_REAL, var, prt1, prt2)
call var_list_set_obs &
    (var_list, var_str ("_User_obs_int"), V_UOBS2_INT, var, prt1, prt2)
end subroutine var_list_set_observables_binary

```

Check if a variable name is defined as an observable:

```

<Variables: procedures>+≡
function string_is_observable_id (string) result (flag)
    logical :: flag
    type(string_t), intent(in) :: string
    select case (char (string))
    case ("PDG", "Hel", "M", "M2", "E", "Px", "Py", "Pz", "P", "P1", "Pt", &
        "Theta", "Phi", "Rap", "Eta", "Theta_RF", "Dist", "kT")
        flag = .true.
    case default
        flag = .false.
    end select
end function string_is_observable_id

```

### 5.5.8 Observables

These are analogous to the unary and binary numeric functions listed above. An observable takes the `pval` component(s) of its one or two argument nodes and produces an integer or real value.

#### Integer-valued unary observables

The PDG code

```
<Variables: procedures>+≡
integer function obs_pdg1 (prt1) result (pdg)
  type(prt_t), intent(in) :: prt1
  pdg = prt_get_pdg (prt1)
end function obs_pdg1
```

The helicity. The return value is meaningful only if the particle is polarized, otherwise an invalid value is returned (-9).

```
<Variables: procedures>+≡
integer function obs_helicity1 (prt1) result (h)
  type(prt_t), intent(in) :: prt1
  if (prt_is_polarized (prt1)) then
    h = prt_get_helicity (prt1)
  else
    h = -9
  end if
end function obs_helicity1
```

#### Real-valued unary observables

The invariant mass squared, obtained from the separately stored value.

```
<Variables: procedures>+≡
real(default) function obs_mass_squared1 (prt1) result (p2)
  type(prt_t), intent(in) :: prt1
  p2 = prt_get_msq (prt1)
end function obs_mass_squared1
```

The signed invariant mass, which is the signed square root of the previous observable.

```
<Variables: procedures>+≡
real(default) function obs_signed_mass1 (prt1) result (m)
  type(prt_t), intent(in) :: prt1
  real(default) :: msq
  msq = prt_get_msq (prt1)
  m = sign (sqrt (abs (msq)), msq)
end function obs_signed_mass1
```

The particle energy

```
<Variables: procedures>+≡
real(default) function obs_energy1 (prt1) result (e)
  type(prt_t), intent(in) :: prt1
```

```

    e = energy (prt_get_momentum (prt1))
end function obs_energy1

```

Particle momentum (components)

*< Variables: procedures >+≡*

```

real(default) function obs_px1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = vector4_get_component (prt_get_momentum (prt1), 1)
end function obs_px1

```

```

real(default) function obs_py1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = vector4_get_component (prt_get_momentum (prt1), 2)
end function obs_py1

```

```

real(default) function obs_pz1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = vector4_get_component (prt_get_momentum (prt1), 3)
end function obs_pz1

```

```

real(default) function obs_p1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = space_part_norm (prt_get_momentum (prt1))
end function obs_p1

```

```

real(default) function obs_pl1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = longitudinal_part (prt_get_momentum (prt1))
end function obs_pl1

```

```

real(default) function obs_pt1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = transverse_part (prt_get_momentum (prt1))
end function obs_pt1

```

Polar and azimuthal angle (lab frame).

*< Variables: procedures >+≡*

```

real(default) function obs_theta1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = polar_angle (prt_get_momentum (prt1))
end function obs_theta1

```

```

real(default) function obs_phi1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = azimuthal_angle (prt_get_momentum (prt1))
end function obs_phi1

```

Rapidity and pseudorapidity

*< Variables: procedures >+≡*

```

real(default) function obs_rap1 (prt1) result (p)
    type(prt_t), intent(in) :: prt1
    p = rapidity (prt_get_momentum (prt1))

```

```

end function obs_rap1

real(default) function obs_eta1 (prt1) result (p)
  type(prt_t), intent(in) :: prt1
  p = pseudorapidity (prt_get_momentum (prt1))
end function obs_eta1

```

Meaningless: Polar angle in the rest frame of the 2nd argument.

```

<Variables: procedures>+=≡
real(default) function obs_theta_rf1 (prt1) result (dist)
  type(prt_t), intent(in) :: prt1
  call msg_fatal (" 'Theta_RF' is undefined as unary observable")
  dist = 0
end function obs_theta_rf1

```

Meaningless: Distance on the  $\eta$ - $\phi$  cylinder.

```

<Variables: procedures>+=≡
real(default) function obs_dist1 (prt1) result (dist)
  type(prt_t), intent(in) :: prt1
  call msg_fatal (" 'Dist' is undefined as unary observable")
  dist = 0
end function obs_dist1

```

### Integer-valued binary observables

These observables are meaningless as binary functions.

```

<Variables: procedures>+=≡
integer function obs_pdg2 (prt1, prt2) result (pdg)
  type(prt_t), intent(in) :: prt1, prt2
  call msg_fatal (" PDG_Code is undefined as binary observable")
  pdg = 0
end function obs_pdg2

integer function obs_helicity2 (prt1, prt2) result (h)
  type(prt_t), intent(in) :: prt1, prt2
  call msg_fatal (" Helicity is undefined as binary observable")
  h = 0
end function obs_helicity2

```

### Real-valued binary observables

The invariant mass squared, obtained from the separately stored value.

```

<Variables: procedures>+=≡
real(default) function obs_mass_squared2 (prt1, prt2) result (p2)
  type(prt_t), intent(in) :: prt1, prt2
  type(prt_t) :: prt
  call prt_init_combine (prt, prt1, prt2)
  p2 = prt_get_msq (prt)
end function obs_mass_squared2

```

The signed invariant mass, which is the signed square root of the previous observable.

```
<Variables: procedures>+≡
  real(default) function obs_signed_mass2 (prt1, prt2) result (m)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    real(default) :: msq
    call prt_init_combine (prt, prt1, prt2)
    msq = prt_get_msq (prt)
    m = sign (sqrt (abs (msq)), msq)
  end function obs_signed_mass2
```

The particle energy

```
<Variables: procedures>+≡
  real(default) function obs_energy2 (prt1, prt2) result (e)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    e = energy (prt_get_momentum (prt))
  end function obs_energy2
```

Particle momentum (components)

```
<Variables: procedures>+≡
  real(default) function obs_px2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = vector4_get_component (prt_get_momentum (prt), 1)
  end function obs_px2

  real(default) function obs_py2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = vector4_get_component (prt_get_momentum (prt), 2)
  end function obs_py2

  real(default) function obs_pz2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = vector4_get_component (prt_get_momentum (prt), 3)
  end function obs_pz2

  real(default) function obs_p2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = space_part_norm (prt_get_momentum (prt))
  end function obs_p2

  real(default) function obs_pl2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
```

```

    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = longitudinal_part (prt_get_momentum (prt))
end function obs_pl2

real(default) function obs_pt2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = transverse_part (prt_get_momentum (prt))
end function obs_pt2

```

Enclosed angle and azimuthal distance (lab frame).

```

<Variables: procedures>+=
real(default) function obs_theta2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    p = enclosed_angle (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_theta2

real(default) function obs_phi2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = azimuthal_distance (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_phi2

```

Rapidity and pseudorapidity distance

```

<Variables: procedures>+=
real(default) function obs_rap2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    p = rapidity_distance &
        (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_rap2

real(default) function obs_eta2 (prt1, prt2) result (p)
    type(prt_t), intent(in) :: prt1, prt2
    type(prt_t) :: prt
    call prt_init_combine (prt, prt1, prt2)
    p = pseudorapidity_distance &
        (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_eta2

```

Polar angle in the rest frame of the 2nd argument.

```

<Variables: procedures>+=
real(default) function obs_theta_rf2 (prt1, prt2) result (theta)
    type(prt_t), intent(in) :: prt1, prt2
    theta = enclosed_angle_rest_frame &
        (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_theta_rf2

```

Distance on the  $\eta$ - $\phi$  cylinder.

```

<Variables: procedures>+=

```

```

real(default) function obs_dist2 (prt1, prt2) result (dist)
  type(prt_t), intent(in) :: prt1, prt2
  dist = eta_phi_distance &
    (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_dist2

```

Durham kT measure.

```

<Variables: procedures>+≡
  real(default) function obs_ktmeasure (prt1, prt2) result (kt)
    type(prt_t), intent(in) :: prt1, prt2
    real (default) :: q2, e1, e2
    !   normalized scale to one for now!
    q2 = 1
    e1 = energy (prt_get_momentum (prt1))
    e2 = energy (prt_get_momentum (prt2))
    kt = (2/q2) * min(e1**2,e2**2) * &
      (1 - enclosed_angle_ct(prt_get_momentum (prt1), &
        prt_get_momentum (prt2)))
  end function obs_ktmeasure

```

### 5.5.9 Event variables

This is a transparent container for the local event variables: weight, squared matrix element. These variables are established as a target within the simulation record, assigned for each event, and can be referenced as local variables inside the event analysis.

```

<Variables: public>+≡
  public :: event_vars_t

<Variables: types>+≡
  type :: event_vars_t
    integer :: event_index = 0
    integer :: process_index = 0
    integer :: process_num_id = 0
    type(string_t) :: process_id
    integer :: n_in = 0
    integer :: n_out = 0
    integer :: n_tot = 0
    real(default) :: sqrts = 0
    real(default) :: sqrts_hat = 0
    real(default) :: sqme = 0
    real(default) :: sqme_ref = 0
    real(default) :: weight = 0
    real(default) :: excess = 0
  end type event_vars_t

<Variables: public>+≡
  public :: event_vars_write

<Variables: procedures>+≡
  subroutine event_vars_write (vars, unit)
    type(event_vars_t), intent(in) :: vars

```



```

integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit)
write (u, *) "Event index          = ", vars%event_index
write (u, *) "Process index        = ", vars%process_index
write (u, *) "Numerical process ID = ", vars%process_num_id
write (u, *) "Process ID           = ", char (vars%process_id)
write (u, *) "Process n_in         = ", vars%n_in
write (u, *) "Process n_out        = ", vars%n_out
write (u, *) "Process n_tot        = ", vars%n_tot
write (u, *) "Event sqrts_hat      = ", vars%sqrts_hat
write (u, *) "Event sqme           = ", vars%sqme
write (u, *) "Event sqme(ref)      = ", vars%sqme_ref
write (u, *) "Event weight         = ", vars%weight
write (u, *) "Event excess weight = ", vars%excess
end subroutine event_vars_write

```

The numerical and string process ID are not read or written in raw form.

*(Variables: public)+≡*

```

public :: event_vars_write_raw
public :: event_vars_read_raw

```

*(Variables: procedures)+≡*

```

subroutine event_vars_write_raw (vars, u, version)
  type(event_vars_t), intent(in) :: vars
  integer, intent(in) :: u
  integer, intent(in) :: version
  write (u) vars%event_index
  write (u) vars%process_index
  select case (version)
  case (3:)
    write (u) vars%n_in
    write (u) vars%n_out
    write (u) vars%n_tot
    write (u) vars%sqrts
    write (u) vars%sqrts_hat
  end select
  write (u) vars%sqme
  write (u) vars%sqme_ref
  write (u) vars%weight
  write (u) vars%excess
end subroutine event_vars_write_raw

subroutine event_vars_read_raw (vars, u, iostat, version)
  type(event_vars_t), intent(out) :: vars
  integer, intent(in) :: u
  integer, intent(out) :: iostat
  integer, intent(in) :: version
  read (u, iostat=iostat) vars%event_index
  if (iostat /= 0) return
  read (u, iostat=iostat) vars%process_index
  if (iostat /= 0) return
  select case (version)
  case (3:)

```

```

        read (u, iostat=iostat)  vars%n_in
        read (u, iostat=iostat)  vars%n_out
        read (u, iostat=iostat)  vars%n_tot
        read (u, iostat=iostat)  vars%sqrts
        read (u, iostat=iostat)  vars%sqrts_hat
    end select
    if (iostat /= 0) return
    read (u, iostat=iostat)  vars%sqme
    if (iostat /= 0) return
    read (u, iostat=iostat)  vars%sqme_ref
    if (iostat /= 0) return
    read (u, iostat=iostat)  vars%weight
    if (iostat /= 0) return
    read (u, iostat=iostat)  vars%excess
end subroutine event_vars_read_raw

```

Append the pointers to the event variables to the local variable list.

*{Variables: public}*+≡

```
public :: var_list_append_event_vars
```

*{Variables: procedures}*+≡

```

subroutine var_list_append_event_vars (var_list, event_vars)
    type(var_list_t), intent(inout) :: var_list
    type(event_vars_t), intent(in), target :: event_vars
    logical, target, save :: known = .true.
    call var_list_append_int_ptr (var_list, &
        var_str ("event_index"), event_vars%event_index, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_int_ptr (var_list, &
        var_str ("process_index"), event_vars%process_index, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_int_ptr (var_list, &
        var_str ("process_num_id"), event_vars%process_num_id, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_string_ptr (var_list, &
        var_str ("process_id"), event_vars%process_id, &      ! $
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_int_ptr (var_list, &
        var_str ("n_in"), event_vars%n_in, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_int_ptr (var_list, &
        var_str ("n_out"), event_vars%n_out, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_int_ptr (var_list, &
        var_str ("n_tot"), event_vars%n_tot, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_real_ptr (var_list, &
        var_str ("sqrts"), event_vars%sqrts, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_real_ptr (var_list, &
        var_str ("sqrts_hat"), event_vars%sqrts_hat, &
        is_known = known, locked = .true., intrinsic = .true.)
    call var_list_append_real_ptr (var_list, &
        var_str ("sqme"), event_vars%sqme, &

```

```

        is_known = known, locked = .true., intrinsic = .true.)
call var_list_append_real_ptr (var_list, &
    var_str ("sqme_ref"), event_vars%sqme, &
    is_known = known, locked = .true., intrinsic = .true.)
call var_list_append_real_ptr (var_list, &
    var_str ("event_weight"), event_vars%weight, &
    is_known = known, locked = .true., intrinsic = .true.)
call var_list_append_real_ptr (var_list, &
    var_str ("event_excess_weight"), event_vars%excess, &
    is_known = known, locked = .true., intrinsic = .true.)
end subroutine var_list_append_event_vars

```

### 5.5.10 API for variable lists

Set a new value. If the variable holds a pointer, this pointer is followed, e.g., a model parameter is actually set. If `ignore` is set, do nothing if the variable does not exist. If `verbose` is set, echo the new value.

The specific versions:

```

{Variables: public}+=
public :: var_list_set_log
public :: var_list_set_int
public :: var_list_set_real
public :: var_list_set_cmplx
public :: var_list_set_subevt
public :: var_list_set_pdg_array
public :: var_list_set_string

{Variables: procedures}+=
subroutine var_list_set_log &
    (var_list, name, lval, is_known, ignore, verbose, model_name)
type(var_list_t), intent(inout), target :: var_list
type(string_t), intent(in) :: name
logical, intent(in) :: lval
logical, intent(in) :: is_known
logical, intent(in), optional :: ignore, verbose
type(string_t), intent(in), optional :: model_name
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr (var_list, name, V_LOG)
if (associated (var)) then
    if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_LOG)
            call var_entry_set_log (var, lval, is_known, verbose, model_name)
        case default
            call var_mismatch_error (name)
        end select
    else
        call var_locked_error (name)
    end if
else
    call var_missing_error (name, ignore)
end if
end subroutine var_list_set_log

```

```

subroutine var_list_set_int &
    (var_list, name, ival, is_known, ignore, verbose, model_name)
    type(var_list_t), intent(inout), target :: var_list
    type(string_t), intent(in) :: name
    integer, intent(in) :: ival
    logical, intent(in) :: is_known
    logical, intent(in), optional :: ignore, verbose
    type(string_t), intent(in), optional :: model_name
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, V_INT)
    if (associated (var)) then
        if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_INT)
                call var_entry_set_int (var, ival, is_known, verbose, model_name)
            case default
                call var_mismatch_error (name)
            end select
        else
            call var_locked_error (name)
        end if
    else
        call var_missing_error (name, ignore)
    end if
end subroutine var_list_set_int

subroutine var_list_set_real &
    (var_list, name, rval, is_known, ignore, verbose, model_name)
    type(var_list_t), intent(inout), target :: var_list
    type(string_t), intent(in) :: name
    real(default), intent(in) :: rval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: ignore, verbose
    type(string_t), intent(in), optional :: model_name
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, V_REAL)
    if (associated (var)) then
        if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_REAL)
                call var_entry_set_real (var, rval, is_known, verbose, model_name)
            case default
                call var_mismatch_error (name)
            end select
        else
            call var_locked_error (name)
        end if
    else
        call var_missing_error (name, ignore)
    end if
end subroutine var_list_set_real

subroutine var_list_set_cmplx &

```

```

        (var_list, name, cval, is_known, ignore, verbose, model_name)
type(var_list_t), intent(inout), target :: var_list
type(string_t), intent(in) :: name
complex(default), intent(in) :: cval
logical, intent(in) :: is_known
logical, intent(in), optional :: ignore, verbose
type(string_t), intent(in), optional :: model_name
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr (var_list, name, V_CMPLX)
if (associated (var)) then
    if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_CMPLX)
            call var_entry_set_cmplx (var, cval, is_known, verbose, model_name)
        case default
            call var_mismatch_error (name)
        end select
    else
        call var_locked_error (name)
    end if
else
    call var_missing_error (name, ignore)
end if
end subroutine var_list_set_cmplx

subroutine var_list_set_pdg_array &
    (var_list, name, aval, is_known, ignore, verbose, model_name)
type(var_list_t), intent(inout), target :: var_list
type(string_t), intent(in) :: name
type(pdg_array_t), intent(in) :: aval
logical, intent(in) :: is_known
logical, intent(in), optional :: ignore, verbose
type(string_t), intent(in), optional :: model_name
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr (var_list, name, V_PDG)
if (associated (var)) then
    if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_PDG)
            call var_entry_set_pdg_array &
                (var, aval, is_known, verbose, model_name)
        case default
            call var_mismatch_error (name)
        end select
    else
        call var_locked_error (name)
    end if
else
    call var_missing_error (name, ignore)
end if
end subroutine var_list_set_pdg_array

subroutine var_list_set_subevt &
    (var_list, name, pval, is_known, ignore, verbose, model_name)

```

```

type(var_list_t), intent(inout), target :: var_list
type(string_t), intent(in) :: name
type(subevt_t), intent(in) :: pval
logical, intent(in) :: is_known
logical, intent(in), optional :: ignore, verbose
type(string_t), intent(in), optional :: model_name
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr (var_list, name, V_SEV)
if (associated (var)) then
    if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_SEV)
            call var_entry_set_subevt &
                (var, pval, is_known, verbose, model_name)
        case default
            call var_mismatch_error (name)
        end select
    else
        call var_locked_error (name)
    end if
else
    call var_missing_error (name, ignore)
end if
end subroutine var_list_set_subevt

subroutine var_list_set_string &
    (var_list, name, sval, is_known, ignore, verbose, model_name)
type(var_list_t), intent(inout), target :: var_list
type(string_t), intent(in) :: name
type(string_t), intent(in) :: sval
logical, intent(in) :: is_known
logical, intent(in), optional :: ignore, verbose
type(string_t), intent(in), optional :: model_name
type(var_entry_t), pointer :: var
var => var_list_get_var_ptr (var_list, name, V_STR)
if (associated (var)) then
    if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_STR)
            call var_entry_set_string &
                (var, sval, is_known, verbose, model_name)
        case default
            call var_mismatch_error (name)
        end select
    else
        call var_locked_error (name)
    end if
else
    call var_missing_error (name, ignore)
end if
end subroutine var_list_set_string

subroutine var_mismatch_error (name)
type(string_t), intent(in) :: name

```

```

        call msg_fatal ("Type mismatch for variable '" // char (name) // "'")
    end subroutine var_mismatch_error

subroutine var_locked_error (name)
    type(string_t), intent(in) :: name
    call msg_error ("Variable '" // char (name) // "' is not user-definable")
end subroutine var_locked_error

subroutine var_missing_error (name, ignore)
    type(string_t), intent(in) :: name
    logical, intent(in), optional :: ignore
    logical :: error
    if (present (ignore)) then
        error = .not. ignore
    else
        error = .true.
    end if
    if (error) then
        call msg_fatal ("Variable '" // char (name) // "' has not been declared")
    end if
end subroutine var_missing_error

```

### 5.5.11 Linking model variables

The variable list of a model can be linked to the global variable list, so the model variables become available. However, the model may change during the execution of the command list, and this is not known at compile time. So, we make a copy of all variables that can be modified by the user; this will include all variables that are present in any of the models. At runtime, the linked list and the pointers to it can be exchanged, but the global variables will stay.

Append a single model variable to the local variable list. The pointer to the original is not set (yet). Check first if it already exists; in that case, do nothing.

```

<Variables: public>+≡
    public :: var_list_init_copy

<Variables: procedures>+≡
    subroutine var_list_init_copy (var_list, model_var, user)
        type(var_list_t), intent(inout), target :: var_list
        type(var_entry_t), intent(in), target :: model_var
        logical, intent(in), optional :: user
        type(var_entry_t), pointer :: var
        if (.not. var_list_exists &
            (var_list, model_var%name, follow_link = .false.)) then
            allocate (var)
            call var_entry_init_copy (var, model_var, user)
            call var_list_append (var_list, var)
        end if
    end subroutine var_list_init_copy

```

Append all model variables, or reuse (unset) them if they already exist. If `derived_only` is set, copy only derived parameters; these are real parameters

that are locked. How should we (should we at all?) generalize this to complex parameters?

```

<Variables: public>+≡
    public :: var_list_init_copies

<Variables: procedures>+≡
    subroutine var_list_init_copies (var_list, model_vars, derived_only)
        type(var_list_t), intent(inout), target :: var_list
        type(var_list_t), intent(in) :: model_vars
        logical, intent(in), optional :: derived_only
        type(var_entry_t), pointer :: model_var, var
        type(string_t) :: name
        logical :: copy_all, locked, derived
        integer :: type
        copy_all = .true.
        if (present (derived_only)) copy_all = .not. derived_only
        model_var => model_vars%first
        do while (associated (model_var))
            name = var_entry_get_name (model_var)
            type = var_entry_get_type (model_var)
            locked = var_entry_is_locked (model_var)
            derived = type == V_REAL .and. locked
            if (copy_all .or. derived) then
                var => var_list_get_var_ptr &
                    (var_list, name, type, follow_link = .false.)
                if (associated (var)) then
                    call var_entry_clear_value (var)
                else
                    allocate (var)
                    call var_entry_init_copy (var, model_var)
                    call var_list_append (var_list, var)
                end if
            end if
            model_var => model_var%next
        end do
    end subroutine var_list_init_copies

```

Clear all previously allocated pointers to some model:

```

<Variables: procedures>+≡
    subroutine var_list_clear_original_pointers (var_list)
        type(var_list_t), intent(inout) :: var_list
        type(var_entry_t), pointer :: var
        var => var_list%first
        do while (associated (var))
            call var_entry_clear_original_pointer (var)
            var => var%next
        end do
    end subroutine var_list_clear_original_pointers

```

Assign the pointer to the original for a single variable.

```

<Variables: public>+≡
    public :: var_list_set_original_pointer

```



```

{Variables: procedures}+=
subroutine var_list_set_original_pointer (var_list, name, model_vars)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: model_vars
  type(var_entry_t), pointer :: var, model_var
  integer :: type
  model_var => var_list_get_var_ptr (model_vars, name)
  if (associated (model_var)) then
    type = var_entry_get_type (model_var)
    var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
    if (associated (var)) then
      call var_entry_set_original_pointer (var, model_var)
    end if
  end if
end subroutine var_list_set_original_pointer

```

Assign pointers to the originals for all variables in the model variable list.

```

{Variables: procedures}+=
subroutine var_list_set_original_pointers (var_list, model_vars)
  type(var_list_t), intent(inout) :: var_list
  type(var_list_t), intent(in), target :: model_vars
  type(var_entry_t), pointer :: var, model_var
  type(string_t) :: name
  integer :: type
  model_var => model_vars%first
  do while (associated (model_var))
    name = var_entry_get_name (model_var)
    type = var_entry_get_type (model_var)
    var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
    if (associated (var)) then
      call var_entry_set_original_pointer (var, model_var)
    end if
    model_var => model_var%next
  end do
end subroutine var_list_set_original_pointers

```

Synchronize the local variable list with the model variable list (which is accessed by the pointers assigned in the previous subroutine).

If `reset_pointers` is unset, do not reset pointers but just update values where a variable is a copy. Resetting pointers is done only in the local variable list.

```

{Variables: public}+=
public :: var_list_synchronize

{Variables: procedures}+=
subroutine var_list_synchronize (var_list, model_vars, reset_pointers)
  type(var_list_t), intent(inout) :: var_list
  type(var_list_t), intent(in), target :: model_vars
  logical, intent(in), optional :: reset_pointers
  type(var_entry_t), pointer :: var
  if (present (reset_pointers)) then
    if (reset_pointers) then

```

```

        call var_list_clear_original_pointers (var_list)
        call var_list_set_original_pointers (var_list, model_vars)
    end if
end if
var => var_list%first
do while (associated (var))
    call var_entry_synchronize (var)
    var => var%next
end do
end subroutine var_list_synchronize

```

This is the inverse operation: synchronize the model variable list with the current variable list. This is necessary after discarding a local variable list.

```

<Variables: public>+≡
    public :: var_list_restore

<Variables: procedures>+≡
    recursive subroutine var_list_restore (var_list)
        type(var_list_t), intent(inout) :: var_list
        type(var_entry_t), pointer :: var
        var => var_list%first
        do while (associated (var))
            call var_entry_restore (var)
            var => var%next
        end do
    end subroutine var_list_restore

```

Mark all entries in the current variable list as undefined. This is done when a local variable list is discarded. If the local list is used again (by a loop), the entries will be re-initialized.

```

<Variables: public>+≡
    public :: var_list_undefine

<Variables: procedures>+≡
    recursive subroutine var_list_undefine (var_list, follow_link)
        type(var_list_t), intent(inout) :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        logical :: rec
        rec = .true.; if (present (follow_link)) rec = follow_link
        var => var_list%first
        do while (associated (var))
            call var_entry_undefine (var)
            var => var%next
        end do
        if (rec .and. associated (var_list%next)) then
            call var_list_undefine (var_list%next, follow_link=follow_link)
        end if
    end subroutine var_list_undefine

```

Make a deep copy of a variable list. The copy does not contain any pointer variables. Clear the original pointer after use, since the original may be lost

when the copy is in use.

```

<Variables: public>+≡
    public :: var_list_init_snapshot

<Variables: procedures>+≡
    recursive subroutine var_list_init_snapshot (var_list, vars_in, follow_link)
        type(var_list_t), intent(out) :: var_list
        type(var_list_t), intent(in) :: vars_in
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var, var_in
        type(var_list_t), pointer :: var_list_next
        logical :: rec
        rec = .true.; if (present (follow_link)) rec = follow_link
        var_in => vars_in%first
        do while (associated (var_in))
            allocate (var)
            call var_entry_init_copy (var, var_in)
            call var_entry_set_original_pointer (var, var_in)
            call var_entry_synchronize (var)
            call var_entry_clear_original_pointer (var)
            call var_list_append (var_list, var)
            var_in => var_in%next
        end do
        if (rec .and. associated (vars_in%next)) then
            allocate (var_list_next)
            call var_list_init_snapshot (var_list_next, vars_in%next)
            call var_list_link (var_list, var_list_next)
        end if
    end subroutine var_list_init_snapshot

```

Check if a user variable can be set. The new flag is set if the user variable has an explicit declaration. If an error occurs, return V\_NONE as variable type.

Also determine the actual type of generic numerical variables, which enter the procedure with type V\_NONE.

```

<Variables: public>+≡
    public :: var_list_check_user_var

<Variables: procedures>+≡
    subroutine var_list_check_user_var (var_list, name, type, new)
        type(var_list_t), intent(in), target :: var_list
        type(string_t), intent(in) :: name
        integer, intent(inout) :: type
        logical, intent(in) :: new
        type(var_entry_t), pointer :: var
        if (string_is_observable_id (name)) then
            call msg_error ("Variable name '" // char (name) &
                // "' is reserved for an observable")
            type = V_NONE
            return
        end if
        if (string_is_integer_result_var (name)) type = V_INT
        var => var_list_get_var_ptr (var_list, name)
        if (associated (var)) then
            if (type == V_NONE) then

```

```

        type = var_entry_get_type (var)
    end if
    if (var_entry_is_locked (var)) then
        call msg_error ("Variable '" // char (name) &
            // "' is not user-definable")
        type = V_NONE
        return
    else if (new) then
        if (var_entry_is_intrinsic (var)) then
            call msg_error ("Intrinsic variable '" &
                // char (name) // "' redeclared")
            type = V_NONE
            return
        end if
        if (var_entry_get_type (var) /= type) then
            call msg_error ("Variable '" // char (name) // "' " &
                // "redeclared with different type")
            type = V_NONE
            return
        end if
    end if
else
    if (string_is_result_var (name)) then
        call msg_error ("Result variable '" // char (name) // "' " &
            // "set without prior integration")
        type = V_NONE
        return
    else if (string_is_num_id (name)) then
        call msg_error ("Numeric process ID '" // char (name) // "' " &
            // "set without process declaration")
        type = V_NONE
        return
    else if (.not. new) then
        call msg_error ("Variable '" // char (name) // "' " &
            // "set without declaration")
        type = V_NONE
        return
    end if
end if
end subroutine var_list_check_user_var

```

Check if a variable name is a result variable of integer type:

*(Variables: procedures)*+≡

```

function string_is_integer_result_var (string) result (flag)
    logical :: flag
    type(string_t), intent(in) :: string
    type(string_t) :: buffer, name, separator
    buffer = string
    call split (buffer, name, "(", separator=separator) ! ")"
    if (separator == "(") then
        select case (char (name))
        case ("num_id", "n_calls")
            flag = .true.

```

```

        case default
            flag = .false.
        end select
    else
        flag = .false.
    end if
end function string_is_integer_result_var

```

Check if a variable name is an integration-result variable:

*(Variables: procedures)+≡*

```

function string_is_result_var (string) result (flag)
    logical :: flag
    type(string_t), intent(in) :: string
    type(string_t) :: buffer, name, separator
    buffer = string
    call split (buffer, name, "(", separator=separator) ! ")"
    if (separator == "(") then
        select case (char (name))
            case ("n_calls", "integral", "error", "accuracy", "chi2", "efficiency")
                flag = .true.
            case default
                flag = .false.
        end select
    else
        flag = .false.
    end if
end function string_is_result_var

```

Check if a variable name is a numeric process ID:

*(Variables: procedures)+≡*

```

function string_is_num_id (string) result (flag)
    logical :: flag
    type(string_t), intent(in) :: string
    type(string_t) :: buffer, name, separator
    buffer = string
    call split (buffer, name, "(", separator=separator) ! ")"
    if (separator == "(") then
        select case (char (name))
            case ("num_id")
                flag = .true.
            case default
                flag = .false.
        end select
    else
        flag = .false.
    end if
end function string_is_num_id

```

## 5.6 Expressions

In this module we define the structures needed to parse a user-defined expression, to compile it into an evaluation tree, and to evaluate it.

We have two flavors of expressions: one with particles and one without particles. The latter version is used for defining cut/selection criteria and for online analysis.

```
<expressions.f90>≡  
  <File header>  
  
  module expressions  
  
    use iso_c_binding !NODEP!  
    <Use kinds>  
    <Use strings>  
    use constants !NODEP!  
    <Use file utils>  
    use diagnostics !NODEP!  
    use lorentz !NODEP!  
    use md5  
    use formats  
    use sorting  
    use ifiles  
    use lexers  
    use syntax_rules  
    use parser  
    use analysis  
    use pdg_arrays  
    use subevents  
    use user_code_interface  
    use variables  
  
    <Standard module head>  
  
    <Expressions: public>  
  
    <Expressions: types>  
  
    <Expressions: interfaces>  
  
    <Expressions: variables>  
  
    contains  
  
    <Expressions: procedures>  
  
  end module expressions
```

### 5.6.1 Tree nodes

The evaluation tree consists of branch nodes (unary and binary) and of leaf nodes, originating from a common root. The node object should be polymorphic. For the time being, polymorphism is emulated here. This means that we

have to maintain all possibilities that the node may hold, including associated procedures as pointers.

The following parameter values characterize the node. Unary and binary operators have sub-nodes. The other are leaf nodes. Possible leafs are literal constants or named-parameter references.

$\langle \text{Expressions: types} \rangle \equiv$

```
integer, parameter :: EN_UNKNOWN = 0, EN_UNARY = 1, EN_BINARY = 2
integer, parameter :: EN_CONSTANT = 3, EN_VARIABLE = 4
integer, parameter :: EN_CONDITIONAL = 5, EN_BLOCK = 6
integer, parameter :: EN_RECORD_CMD = 7
integer, parameter :: EN_OBS1_INT = 11, EN_OBS2_INT = 12
integer, parameter :: EN_OBS1_REAL = 21, EN_OBS2_REAL = 22
integer, parameter :: EN_UOBS1_INT = 31, EN_UOBS2_INT = 32
integer, parameter :: EN_UOBS1_REAL = 41, EN_UOBS2_REAL = 42
integer, parameter :: EN_PRT_FUN_UNARY = 101, EN_PRT_FUN_BINARY = 102
integer, parameter :: EN_EVAL_FUN_UNARY = 111, EN_EVAL_FUN_BINARY = 112
integer, parameter :: EN_LOG_FUN_UNARY = 121, EN_LOG_FUN_BINARY = 122
integer, parameter :: EN_INT_FUN_UNARY = 131, EN_INT_FUN_BINARY = 132
integer, parameter :: EN_REAL_FUN_UNARY = 141, EN_REAL_FUN_BINARY = 142
integer, parameter :: EN_FORMAT_STR = 161
```

$\langle \text{Expressions: types} \rangle + \equiv$

```
type :: eval_node_t
private
type(string_t) :: tag
integer :: type = EN_UNKNOWN
integer :: result_type = V_NONE
type(var_list_t), pointer :: var_list => null ()
type(string_t) :: var_name
logical, pointer :: value_is_known => null ()
logical, pointer :: lval => null ()
integer, pointer :: ival => null ()
real(default), pointer :: rval => null ()
complex(default), pointer :: cval => null ()
type(subevt_t), pointer :: pval => null ()
type(pdg_array_t), pointer :: aval => null ()
type(string_t), pointer :: sval => null ()
type(eval_node_t), pointer :: arg0 => null ()
type(eval_node_t), pointer :: arg1 => null ()
type(eval_node_t), pointer :: arg2 => null ()
type(eval_node_t), pointer :: arg3 => null ()
type(eval_node_t), pointer :: arg4 => null ()
procedure(obs_unary_int), nopass, pointer :: obs1_int => null ()
procedure(obs_unary_real), nopass, pointer :: obs1_real => null ()
procedure(obs_binary_int), nopass, pointer :: obs2_int => null ()
procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
integer, pointer :: prt_type => null ()
integer, pointer :: index => null ()
real(default), pointer :: tolerance => null ()
type(prt_t), pointer :: prt1 => null ()
type(prt_t), pointer :: prt2 => null ()
procedure(unary_log), nopass, pointer :: op1_log => null ()
procedure(unary_int), nopass, pointer :: op1_int => null ()
```

```

procedure(unary_real), nopass, pointer :: op1_real => null ()
procedure(unary_cmplx), nopass, pointer :: op1_cmplx => null ()
procedure(unary_pdg), nopass, pointer :: op1_pdg => null ()
procedure(unary_sev), nopass, pointer :: op1_sev => null ()
procedure(unary_str), nopass, pointer :: op1_str => null ()
procedure(unary_cut), nopass, pointer :: op1_cut => null ()
procedure(unary_evi), nopass, pointer :: op1_evi => null ()
procedure(unary_evr), nopass, pointer :: op1_evr => null ()
procedure(binary_log), nopass, pointer :: op2_log => null ()
procedure(binary_int), nopass, pointer :: op2_int => null ()
procedure(binary_real), nopass, pointer :: op2_real => null ()
procedure(binary_cmplx), nopass, pointer :: op2_cmplx => null ()
procedure(binary_pdg), nopass, pointer :: op2_pdg => null ()
procedure(binary_sev), nopass, pointer :: op2_sev => null ()
procedure(binary_str), nopass, pointer :: op2_str => null ()
procedure(binary_cut), nopass, pointer :: op2_cut => null ()
procedure(binary_evi), nopass, pointer :: op2_evi => null ()
procedure(binary_evr), nopass, pointer :: op2_evr => null ()
end type eval_node_t

```

Finalize a node recursively. Allocated constants are deleted, pointers are ignored.

*(Expressions: procedures)*≡

```

recursive subroutine eval_node_final_rec (node)
  type(eval_node_t), intent(inout) :: node
  select case (node%type)
  case (EN_UNARY)
    call eval_node_final_rec (node%arg1)
  case (EN_BINARY)
    call eval_node_final_rec (node%arg1)
    call eval_node_final_rec (node%arg2)
  case (EN_CONDITIONAL)
    call eval_node_final_rec (node%arg0)
    call eval_node_final_rec (node%arg1)
    call eval_node_final_rec (node%arg2)
  case (EN_BLOCK)
    call eval_node_final_rec (node%arg0)
    call eval_node_final_rec (node%arg1)
  case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
        EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
    if (associated (node%arg0)) call eval_node_final_rec (node%arg0)
    call eval_node_final_rec (node%arg1)
    deallocate (node%index)
    deallocate (node%prt1)
  case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
        EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
    if (associated (node%arg0)) call eval_node_final_rec (node%arg0)
    call eval_node_final_rec (node%arg1)
    call eval_node_final_rec (node%arg2)
    deallocate (node%index)
    deallocate (node%prt1)
    deallocate (node%prt2)
  case (EN_FORMAT_STR)

```



```

        if (associated (node%arg0)) call eval_node_final_rec (node%arg0)
        if (associated (node%arg1)) call eval_node_final_rec (node%arg1)
        deallocate (node%ival)
    case (EN_RECORD_CMD)
        if (associated (node%arg0)) call eval_node_final_rec (node%arg0)
        if (associated (node%arg1)) call eval_node_final_rec (node%arg1)
        if (associated (node%arg2)) call eval_node_final_rec (node%arg2)
        if (associated (node%arg3)) call eval_node_final_rec (node%arg3)
        if (associated (node%arg4)) call eval_node_final_rec (node%arg4)
    end select
    select case (node%type)
    case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, EN_CONSTANT, EN_BLOCK, &
        EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
        EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
        EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
        EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
        EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY, &
        EN_FORMAT_STR, EN_RECORD_CMD)
        select case (node%result_type)
        case (V_LOG); deallocate (node%lval)
        case (V_INT); deallocate (node%ival)
        case (V_REAL); deallocate (node%rval)
        case (V_CMPLX); deallocate (node%cval)
        case (V_SEV); deallocate (node%pval)
        case (V_PDG); deallocate (node%aval)
        case (V_STR); deallocate (node%sval)
        end select
        deallocate (node%value_is_known)
    end select
end subroutine eval_node_final_rec

```

## Leaf nodes

Initialize a leaf node with a literal constant.

*(Expressions: procedures)* +=

```

subroutine eval_node_init_log (node, lval)
    type(eval_node_t), intent(out) :: node
    logical, intent(in) :: lval
    node%type = EN_CONSTANT
    node%result_type = V_LOG
    allocate (node%lval, node%value_is_known)
    node%lval = lval
    node%value_is_known = .true.
end subroutine eval_node_init_log

subroutine eval_node_init_int (node, ival)
    type(eval_node_t), intent(out) :: node
    integer, intent(in) :: ival
    node%type = EN_CONSTANT
    node%result_type = V_INT
    allocate (node%ival, node%value_is_known)
    node%ival = ival
    node%value_is_known = .true.

```

```

end subroutine eval_node_init_int

subroutine eval_node_init_real (node, rval)
  type(eval_node_t), intent(out) :: node
  real(default), intent(in) :: rval
  node%type = EN_CONSTANT
  node%result_type = V_REAL
  allocate (node%rval, node%value_is_known)
  node%rval = rval
  node%value_is_known = .true.
end subroutine eval_node_init_real

subroutine eval_node_init_cmplx (node, cval)
  type(eval_node_t), intent(out) :: node
  complex(default), intent(in) :: cval
  node%type = EN_CONSTANT
  node%result_type = V_CMPLX
  allocate (node%cval, node%value_is_known)
  node%cval = cval
  node%value_is_known = .true.
end subroutine eval_node_init_cmplx

subroutine eval_node_init_subevt (node, pval)
  type(eval_node_t), intent(out) :: node
  type(subevt_t), intent(in) :: pval
  node%type = EN_CONSTANT
  node%result_type = V_SEV
  allocate (node%pval, node%value_is_known)
  node%pval = pval
  node%value_is_known = .true.
end subroutine eval_node_init_subevt

subroutine eval_node_init_pdg_array (node, aval)
  type(eval_node_t), intent(out) :: node
  type(pdg_array_t), intent(in) :: aval
  node%type = EN_CONSTANT
  node%result_type = V_PDG
  allocate (node%aval, node%value_is_known)
  node%aval = aval
  node%value_is_known = .true.
end subroutine eval_node_init_pdg_array

subroutine eval_node_init_string (node, sval)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: sval
  node%type = EN_CONSTANT
  node%result_type = V_STR
  allocate (node%sval, node%value_is_known)
  node%sval = sval
  node%value_is_known = .true.
end subroutine eval_node_init_string

```

Initialize a leaf node with a pointer to a named parameter  
*(Expressions: procedures)* +  $\equiv$

```

subroutine eval_node_init_log_ptr (node, name, lval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  logical, intent(in), target :: lval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_LOG
  node%lval => lval
  node%value_is_known => is_known
end subroutine eval_node_init_log_ptr

subroutine eval_node_init_int_ptr (node, name, ival, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  integer, intent(in), target :: ival
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_INT
  node%ival => ival
  node%value_is_known => is_known
end subroutine eval_node_init_int_ptr

subroutine eval_node_init_real_ptr (node, name, rval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  real(default), intent(in), target :: rval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_REAL
  node%rval => rval
  node%value_is_known => is_known
end subroutine eval_node_init_real_ptr

subroutine eval_node_init_cmplx_ptr (node, name, cval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  complex(default), intent(in), target :: cval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_CMPLX
  node%cval => cval
  node%value_is_known => is_known
end subroutine eval_node_init_cmplx_ptr

subroutine eval_node_init_subevt_ptr (node, name, pval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(subevt_t), intent(in), target :: pval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE

```

```

    node%tag = name
    node%result_type = V_SEV
    node%pval => pval
    node%value_is_known => is_known
end subroutine eval_node_init_subevt_ptr

subroutine eval_node_init_pdg_array_ptr (node, name, aval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_PDG
    node%aval => aval
    node%value_is_known => is_known
end subroutine eval_node_init_pdg_array_ptr

subroutine eval_node_init_string_ptr (node, name, sval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_STR
    node%sval => sval
    node%value_is_known => is_known
end subroutine eval_node_init_string_ptr

```

## Branch nodes

Initialize a branch node, sub-nodes are given.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_branch (node, tag, result_type, arg1, arg2)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: tag
    integer, intent(in) :: result_type
    type(eval_node_t), intent(in), target :: arg1
    type(eval_node_t), intent(in), target, optional :: arg2
    if (present (arg2)) then
        node%type = EN_BINARY
    else
        node%type = EN_UNARY
    end if
    node%tag = tag
    node%result_type = result_type
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    if (present (arg2)) node%arg2 => arg2
end subroutine eval_node_init_branch

```

Allocate the node value according to the result type.

*(Expressions: procedures)* +≡

```

subroutine eval_node_allocate_value (node)
  type(eval_node_t), intent(inout) :: node
  select case (node%result_type)
    case (V_LOG); allocate (node%lval)
    case (V_INT); allocate (node%ival)
    case (V_REAL); allocate (node%rval)
    case (V_CMPLX); allocate (node%cval)
    case (V_PDG); allocate (node%aval)
    case (V_SEV); allocate (node%pval)
      call subevt_init (node%pval)
    case (V_STR); allocate (node%sval)
  end select
  allocate (node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_allocate_value

```

Initialize a node with an observable. This includes a procedure pointer. The input is a variable entry from the stack which holds the necessary information.

The user-observable version does not contain a pointer. The appropriate procedure is called directly at evaluation time. Furthermore, this is a branch node with a reference to the string-expression node which determines the name of the user procedure.

*(Expressions: procedures)* +≡

```

subroutine eval_node_init_obs (node, var, arg)
  type(eval_node_t), intent(out) :: node
  type(var_entry_t), intent(in), target :: var
  type(eval_node_t), intent(in), target, optional :: arg
  integer :: var_type
  node%tag = var_entry_get_name (var)
  var_type = var_entry_get_type (var)
  select case (var_type)
    case (V_OBS1_INT)
      node%type = EN_OBS1_INT
      call var_entry_assign_obs1_int_ptr (node%obs1_int, var)
    case (V_OBS2_INT)
      node%type = EN_OBS2_INT
      call var_entry_assign_obs2_int_ptr (node%obs2_int, var)
    case (V_OBS1_REAL)
      node%type = EN_OBS1_REAL
      call var_entry_assign_obs1_real_ptr (node%obs1_real, var)
    case (V_OBS2_REAL)
      node%type = EN_OBS2_REAL
      call var_entry_assign_obs2_real_ptr (node%obs2_real, var)
    case (V_UOBS1_INT)
      node%type = EN_UOBS1_INT
    case (V_UOBS2_INT)
      node%type = EN_UOBS2_INT
    case (V_UOBS1_REAL)
      node%type = EN_UOBS1_REAL
    case (V_UOBS2_REAL)
      node%type = EN_UOBS2_REAL
  end select
end subroutine eval_node_init_obs

```

```

end select
select case (var_type)
case (V_OBS1_INT, V_OBS2_INT, V_UOBS1_INT, V_UOBS2_INT)
    node%result_type = V_INT
    allocate (node%ival, node%value_is_known)
    node%value_is_known = .false.
case (V_OBS1_REAL, V_OBS2_REAL, V_UOBS1_REAL, V_UOBS2_REAL)
    node%result_type = V_REAL
    allocate (node%rval, node%value_is_known)
    node%value_is_known = .false.
end select
select case (var_type)
case (V_OBS1_INT, V_OBS1_REAL, V_UOBS1_INT, V_UOBS1_REAL)
    node%prt1 => var_entry_get_prt1_ptr (var)
case (V_OBS2_INT, V_OBS2_REAL, V_UOBS2_INT, V_UOBS2_REAL)
    node%prt1 => var_entry_get_prt1_ptr (var)
    node%prt2 => var_entry_get_prt2_ptr (var)
end select
select case (var_type)
case (V_UOBS1_INT, V_UOBS1_REAL, V_UOBS2_INT, V_UOBS2_REAL)
    if (present (arg)) node%arg0 => arg
end select
end subroutine eval_node_init_obs

```

Initialize a block node which contains, in addition to the expression to be evaluated, a variable definition. The result type is not yet assigned, because we can compile the enclosed expression only after the var list is set up.

Note that the node always allocates a new variable list and appends it to the current one. Thus, if the variable redefines an existing one, it only shadows it but does not reset it. Any side-effects are therefore absent and need not be undone outside the block.

If the flag `new` is set, a variable is (re)declared. This must not be done for intrinsic variables. Vice versa, if the variable is not existent, the `new` flag is required.

*(Expressions: procedures)+≡*

```

subroutine eval_node_init_block (node, name, type, var_def, var_list)
    type(eval_node_t), intent(out), target :: node
    type(string_t), intent(in) :: name
    integer, intent(in) :: type
    type(eval_node_t), intent(in), target :: var_def
    type(var_list_t), intent(in), target :: var_list
    node%type = EN_BLOCK
    node%tag = "var_def"
    node%var_name = name
    node%arg1 => var_def
    allocate (node%var_list)
    call var_list_link (node%var_list, var_list)
    if (var_def%type == EN_CONSTANT) then
        select case (type)
        case (V_LOG)
            call var_list_append_log (node%var_list, name, var_def%lval)
        case (V_INT)
            call var_list_append_int (node%var_list, name, var_def%ival)

```

```

case (V_REAL)
  call var_list_append_real (node%var_list, name, var_def%rval)
case (V_CMPLX)
  call var_list_append_cmplx (node%var_list, name, var_def%cval)
case (V_PDG)
  call var_list_append_pdg_array &
    (node%var_list, name, var_def%aval)
case (V_SEV)
  call var_list_append_subevt &
    (node%var_list, name, var_def%pval)
case (V_STR)
  call var_list_append_string (node%var_list, name, var_def%sval)
end select
else
  select case (type)
  case (V_LOG); call var_list_append_log_ptr &
    (node%var_list, name, var_def%lval, var_def%value_is_known)
  case (V_INT); call var_list_append_int_ptr &
    (node%var_list, name, var_def%ival, var_def%value_is_known)
  case (V_REAL); call var_list_append_real_ptr &
    (node%var_list, name, var_def%rval, var_def%value_is_known)
  case (V_CMPLX); call var_list_append_cmplx_ptr &
    (node%var_list, name, var_def%cval, var_def%value_is_known)
  case (V_PDG); call var_list_append_pdg_array_ptr &
    (node%var_list, name, var_def%aval, var_def%value_is_known)
  case (V_SEV); call var_list_append_subevt_ptr &
    (node%var_list, name, var_def%pval, var_def%value_is_known)
  case (V_STR); call var_list_append_string_ptr &
    (node%var_list, name, var_def%sval, var_def%value_is_known)
  end select
end if
end subroutine eval_node_init_block

```

Complete block initialization by assigning the expression to evaluate to `arg0`.

*(Expressions: procedures)*+≡

```

subroutine eval_node_set_expr (node, arg, result_type)
  type(eval_node_t), intent(inout) :: node
  type(eval_node_t), intent(in), target :: arg
  integer, intent(in), optional :: result_type
  if (present (result_type)) then
    node%result_type = result_type
  else
    node%result_type = arg%result_type
  end if
  call eval_node_allocate_value (node)
  node%arg0 => arg
end subroutine eval_node_set_expr

```

Initialize a conditional. There are three branches: the condition (evaluates to logical) and the two alternatives (evaluate both to the same arbitrary type).

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_conditional (node, result_type, cond, arg1, arg2)
  type(eval_node_t), intent(out) :: node

```

```

integer, intent(in) :: result_type
type(eval_node_t), intent(in), target :: cond, arg1, arg2
node%type = EN_CONDITIONAL
node%tag = "cond"
node%result_type = result_type
call eval_node_allocate_value (node)
node%arg0 => cond
node%arg1 => arg1
node%arg2 => arg2
end subroutine eval_node_init_conditional

```

Initialize a recording command (which evaluates to a logical constant). The first branch is the ID of the analysis object to be filled, the optional branches 1 to 4 are the values to be recorded.

If the event-weight pointer is null, we record values with unit weight. Otherwise, we use the value pointed to as event weight.

There can be up to four arguments which represent  $x$ ,  $y$ ,  $\Delta y$ ,  $\Delta x$ . Therefore, this is the only node type that may fill four sub-nodes.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_record_cmd &
  (node, event_weight, id, arg1, arg2, arg3, arg4)
type(eval_node_t), intent(out) :: node
real(default), pointer :: event_weight
type(eval_node_t), intent(in), target :: id
type(eval_node_t), intent(in), optional, target :: arg1, arg2, arg3, arg4
call eval_node_init_log (node, .true.)
node%type = EN_RECORD_CMD
node%rval => event_weight
node%tag = "record_cmd"
node%arg0 => id
if (present (arg1)) then
  node%arg1 => arg1
  if (present (arg2)) then
    node%arg2 => arg2
    if (present (arg3)) then
      node%arg3 => arg3
      if (present (arg4)) then
        node%arg4 => arg4
      end if
    end if
  end if
end if
end if
end if
end subroutine eval_node_init_record_cmd

```

Initialize a node for operations on subevents. The particle lists (one or two) are inserted as **arg1** and **arg2**. We allocated particle pointers as temporaries for iterating over particle lists. The procedure pointer which holds the function to evaluate for the subevents (e.g., combine, select) is also initialized.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_prt_fun_unary (node, arg1, name, proc)
type(eval_node_t), intent(out) :: node
type(eval_node_t), intent(in), target :: arg1

```



```

    type(string_t), intent(in) :: name
    procedure(unary_sev) :: proc
    node%type = EN_PRT_FUN_UNARY
    node%tag = name
    node%result_type = V_SEV
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index)
    allocate (node%prt1)
    node%op1_sev => proc
end subroutine eval_node_init_prt_fun_unary

subroutine eval_node_init_prt_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_sev) :: proc
    node%type = EN_PRT_FUN_BINARY
    node%tag = name
    node%result_type = V_SEV
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_sev => proc
end subroutine eval_node_init_prt_fun_binary

```

Similar, but for particle-list functions that evaluate to a real value.

(*Expressions: procedures*)+≡

```

subroutine eval_node_init_eval_fun_unary (node, arg1, name)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    node%type = EN_EVAL_FUN_UNARY
    node%tag = name
    node%result_type = V_REAL
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index)
    allocate (node%prt1)
end subroutine eval_node_init_eval_fun_unary

subroutine eval_node_init_eval_fun_binary (node, arg1, arg2, name)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    node%type = EN_EVAL_FUN_BINARY
    node%tag = name
    node%result_type = V_REAL
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2

```

```

        allocate (node%index)
        allocate (node%prt1)
        allocate (node%prt2)
    end subroutine eval_node_init_eval_fun_binary

```

These are for particle-list functions that evaluate to a logical value.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_log_fun_unary (node, arg1, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    procedure(unary_cut) :: proc
    node%type = EN_LOG_FUN_UNARY
    node%tag = name
    node%result_type = V_LOG
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index)
    allocate (node%prt1)
    node%op1_cut => proc
end subroutine eval_node_init_log_fun_unary

subroutine eval_node_init_log_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_cut) :: proc
    node%type = EN_LOG_FUN_BINARY
    node%tag = name
    node%result_type = V_LOG
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_cut => proc
end subroutine eval_node_init_log_fun_binary

```

These are for particle-list functions that evaluate to an integer value.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_int_fun_unary (node, arg1, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    procedure(unary_evi) :: proc
    node%type = EN_INT_FUN_UNARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index)
    allocate (node%prt1)

```

```

        node%op1_evi => proc
end subroutine eval_node_init_int_fun_unary

subroutine eval_node_init_int_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_evi) :: proc
    node%type = EN_INT_FUN_BINARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_evi => proc
end subroutine eval_node_init_int_fun_binary

```

These are for particle-list functions that evaluate to a real value.

*(Expressions: procedures)*+≡

```

subroutine eval_node_init_real_fun_unary (node, arg1, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    procedure(unary_evr) :: proc
    node%type = EN_REAL_FUN_UNARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index)
    allocate (node%prt1)
    node%op1_evr => proc
end subroutine eval_node_init_real_fun_unary

subroutine eval_node_init_real_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_evr) :: proc
    node%type = EN_REAL_FUN_BINARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_evr => proc
end subroutine eval_node_init_real_fun_binary

```

Initialize a node for a string formatting function (sprintf).

```

(Expressions: procedures)+≡
subroutine eval_node_init_format_string (node, fmt, arg, name, n_args)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), pointer :: fmt, arg
  type(string_t), intent(in) :: name
  integer, intent(in) :: n_args
  node%type = EN_FORMAT_STR
  node%tag = name
  node%result_type = V_STR
  call eval_node_allocate_value (node)
  node%arg0 => fmt
  node%arg1 => arg
  allocate (node%ival)
  node%ival = n_args
end subroutine eval_node_init_format_string

```

If particle functions depend upon a condition (or an expression is evaluated), the observables that can be evaluated for the given particles have to be thrown on the local variable stack. This is done here. Each observable is initialized with the particle pointers which have been allocated for the node.

The integer variable that is referred to by the Index pseudo-observable is always known when it is referred to.

```

(Expressions: procedures)+≡
subroutine eval_node_set_observables (node, var_list)
  type(eval_node_t), intent(inout) :: node
  type(var_list_t), intent(in), target :: var_list
  logical, save, target :: known = .true.
  allocate (node%var_list)
  call var_list_link (node%var_list, var_list)
  allocate (node%index)
  call var_list_append_int_ptr &
    (node%var_list, var_str ("Index"), node%index, known, intrinsic=.true.)
  if (.not. associated (node%prt2)) then
    call var_list_set_observables_unary &
      (node%var_list, node%prt1)
  else
    call var_list_set_observables_binary &
      (node%var_list, node%prt1, node%prt2)
  end if
end subroutine eval_node_set_observables

```

## Output

```

(Expressions: procedures)+≡
subroutine eval_node_write (node, unit, indent)
  type(eval_node_t), intent(in) :: node
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = output_unit (unit); if (u < 0) return
  ind = 0; if (present (indent)) ind = indent

```

```

write (u, "(A)", advance="no") repeat ("| ", ind) // "o "
select case (node%type)
case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, &
      EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
      EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
      EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
      EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
      EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY)
  write (u, "(A)", advance="no") "[" // char (node%tag) // "]" = "
case (EN_CONSTANT)
  write (u, "(A)", advance="no") "[const] ="
case (EN_VARIABLE)
  write (u, "(A)", advance="no") char (node%tag) // " =>"
case (EN_OBS1_INT, EN_OBS2_INT, EN_OBS1_REAL, EN_OBS2_REAL, &
      EN_UOBS1_INT, EN_UOBS2_INT, EN_UOBS1_REAL, EN_UOBS2_REAL)
  write (u, "(A)", advance="no") char (node%tag) // " ="
case (EN_BLOCK)
  write (u, "(A)", advance="no") "[" // char (node%tag) // "]" // &
    char (node%var_name) // " [expr] = "
case default
  write (u, "(A)", advance="no") "[???] ="
end select
select case (node%result_type)
case (V_LOG)
  if (node%value_is_known) then
    if (node%lval) then
      write (u, "(1x,A)") "true"
    else
      write (u, "(1x,A)") "false"
    end if
  else
    write (u, "(1x,A)") "[unknown logical]"
  end if
case (V_INT)
  if (node%value_is_known) then
    write (u, "(1x,I0)") node%ival
  else
    write (u, "(1x,A)") "[unknown integer]"
  end if
case (V_REAL)
  if (node%value_is_known) then
    write (u, "(1x,ES19.12)") node%rval
  else
    write (u, "(1x,A)") "[unknown real]"
  end if
case (V_CMPLX)
  if (node%value_is_known) then
    write (u, "(1x,'(,ES19.12,',',ES19.12,')')") node%cval
  else
    write (u, "(1x,A)") "[unknown complex]"
  end if
case (V_SEV)
  if (char (node%tag) == "@evt") then
    write (u, "(1x,A)") "[event subevent]"

```

```

        else if (node%value_is_known) then
            call subevt_write &
                (node%pval, unit, prefix = repeat ("| ", ind + 1))
        else
            write (u, "(1x,A)") "[unknown subevent]"
        end if
    case (V_PDG)
        write (u, "(1x)", advance="no")
        call pdg_array_write (node%aval, u); write (u, *)
    case (V_STR)
        if (node%value_is_known) then
            write (u, "(A)") ''' // char (node%sval) // '''
        else
            write (u, "(1x,A)") "[unknown string]"
        end if
    case default
        write (u, "(1x,A)") "[empty]"
    end select
select case (node%type)
case (EN_OBS1_INT, EN_OBS1_REAL, EN_UOBS1_INT, EN_UOBS1_REAL)
    write (u, "(A,6x,A)", advance="no") repeat ("| ", ind), "prt1 ="
    call prt_write (node%prt1, unit)
case (EN_OBS2_INT, EN_OBS2_REAL, EN_UOBS2_INT, EN_UOBS2_REAL)
    write (u, "(A,6x,A)", advance="no") repeat ("| ", ind), "prt1 ="
    call prt_write (node%prt1, unit)
    write (u, "(A,6x,A)", advance="no") repeat ("| ", ind), "prt2 ="
    call prt_write (node%prt2, unit)
end select
end subroutine eval_node_write

recursive subroutine eval_node_write_rec (node, unit, indent)
    type(eval_node_t), intent(in) :: node
    integer, intent(in), optional :: unit
    integer, intent(in), optional :: indent
    integer :: u, ind
    u = output_unit (unit); if (u < 0) return
    ind = 0; if (present (indent)) ind = indent
    call eval_node_write (node, unit, indent)
    select case (node%type)
    case (EN_UNARY)
        if (associated (node%arg0)) &
            call eval_node_write_rec (node%arg0, unit, ind+1)
        call eval_node_write_rec (node%arg1, unit, ind+1)
    case (EN_BINARY)
        if (associated (node%arg0)) &
            call eval_node_write_rec (node%arg0, unit, ind+1)
        call eval_node_write_rec (node%arg1, unit, ind+1)
        call eval_node_write_rec (node%arg2, unit, ind+1)
    case (EN_BLOCK)
        call eval_node_write_rec (node%arg1, unit, ind+1)
        call eval_node_write_rec (node%arg0, unit, ind+1)
    case (EN_CONDITIONAL)
        call eval_node_write_rec (node%arg0, unit, ind+1)
        call eval_node_write_rec (node%arg1, unit, ind+1)

```

```

        call eval_node_write_rec (node%arg2, unit, ind+1)
    case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
          EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
        if (associated (node%arg0)) &
            call eval_node_write_rec (node%arg0, unit, ind+1)
        call eval_node_write_rec (node%arg1, unit, ind+1)
    case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
          EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
        if (associated (node%arg0)) &
            call eval_node_write_rec (node%arg0, unit, ind+1)
        call eval_node_write_rec (node%arg1, unit, ind+1)
        call eval_node_write_rec (node%arg2, unit, ind+1)
    end select
end subroutine eval_node_write_rec

```

## 5.6.2 Operation types

For the operations associated to evaluation tree nodes, we define abstract interfaces for all cases.

Particles/subevents are transferred by-reference, to avoid unnecessary copying. Therefore, subroutines instead of functions. (Furthermore, the function version of `unary_prt` triggers an obscure bug in nagfor 5.2(649) [invalid C code].)

*(Expressions: interfaces)*≡

```

abstract interface
    logical function unary_log (arg)
        import eval_node_t
        type(eval_node_t), intent(in) :: arg
    end function unary_log
end interface
abstract interface
    integer function unary_int (arg)
        import eval_node_t
        type(eval_node_t), intent(in) :: arg
    end function unary_int
end interface
abstract interface
    real(default) function unary_real (arg)
        import default
        import eval_node_t
        type(eval_node_t), intent(in) :: arg
    end function unary_real
end interface
abstract interface
    complex(default) function unary_cmplx (arg)
        import default
        import eval_node_t
        type(eval_node_t), intent(in) :: arg
    end function unary_cmplx
end interface
abstract interface
    subroutine unary_pdg (pdg_array, arg)
        import pdg_array_t

```

```

        import eval_node_t
        type(pdg_array_t), intent(out) :: pdg_array
        type(eval_node_t), intent(in) :: arg
    end subroutine unary_pdg
end interface
abstract interface
    subroutine unary_sev (subevt, arg, arg0)
        import subevt_t
        import eval_node_t
        type(subevt_t), intent(inout) :: subevt
        type(eval_node_t), intent(in) :: arg
        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine unary_sev
end interface
abstract interface
    subroutine unary_str (string, arg)
        import string_t
        import eval_node_t
        type(string_t), intent(out) :: string
        type(eval_node_t), intent(in) :: arg
    end subroutine unary_str
end interface
abstract interface
    logical function unary_cut (arg1, arg0)
        import eval_node_t
        type(eval_node_t), intent(in) :: arg1
        type(eval_node_t), intent(inout) :: arg0
    end function unary_cut
end interface
abstract interface
    subroutine unary_evi (ival, arg1, arg0)
        import eval_node_t
        integer, intent(out) :: ival
        type(eval_node_t), intent(in) :: arg1
        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine unary_evi
end interface
abstract interface
    subroutine unary_evr (rval, arg1, arg0)
        import eval_node_t, default
        real(default), intent(out) :: rval
        type(eval_node_t), intent(in) :: arg1
        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine unary_evr
end interface
abstract interface
    logical function binary_log (arg1, arg2)
        import eval_node_t
        type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_log
end interface
abstract interface
    integer function binary_int (arg1, arg2)
        import eval_node_t

```



```

        type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_int
end interface
abstract interface
    real(default) function binary_real (arg1, arg2)
        import default
        import eval_node_t
        type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_real
end interface
abstract interface
    complex(default) function binary_cmplx (arg1, arg2)
        import default
        import eval_node_t
        type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_cmplx
end interface
abstract interface
    subroutine binary_pdg (pdg_array, arg1, arg2)
        import pdg_array_t
        import eval_node_t
        type(pdg_array_t), intent(out) :: pdg_array
        type(eval_node_t), intent(in) :: arg1, arg2
    end subroutine binary_pdg
end interface
abstract interface
    subroutine binary_sev (subevt, arg1, arg2, arg0)
        import subevt_t
        import eval_node_t
        type(subevt_t), intent(inout) :: subevt
        type(eval_node_t), intent(in) :: arg1, arg2
        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine binary_sev
end interface
abstract interface
    subroutine binary_str (string, arg1, arg2)
        import string_t
        import eval_node_t
        type(string_t), intent(out) :: string
        type(eval_node_t), intent(in) :: arg1, arg2
    end subroutine binary_str
end interface
abstract interface
    logical function binary_cut (arg1, arg2, arg0)
        import eval_node_t
        type(eval_node_t), intent(in) :: arg1, arg2
        type(eval_node_t), intent(inout) :: arg0
    end function binary_cut
end interface
abstract interface
    subroutine binary_evi (ival, arg1, arg2, arg0)
        import eval_node_t
        integer, intent(out) :: ival
        type(eval_node_t), intent(in) :: arg1, arg2
    end subroutine binary_evi
end interface

```

```

        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine binary_evi
end interface
abstract interface
    subroutine binary_evr (rval, arg1, arg2, arg0)
        import eval_node_t, default
        real(default), intent(out) :: rval
        type(eval_node_t), intent(in) :: arg1, arg2
        type(eval_node_t), intent(inout), optional :: arg0
    end subroutine binary_evr
end interface

```

The following subroutines set the procedure pointer:

(*Expressions: procedures*) +=

```

subroutine eval_node_set_op1_log (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_log) :: op
    en%op1_log => op
end subroutine eval_node_set_op1_log

subroutine eval_node_set_op1_int (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_int) :: op
    en%op1_int => op
end subroutine eval_node_set_op1_int

subroutine eval_node_set_op1_real (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_real) :: op
    en%op1_real => op
end subroutine eval_node_set_op1_real

subroutine eval_node_set_op1_cmplx (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_cmplx) :: op
    en%op1_cmplx => op
end subroutine eval_node_set_op1_cmplx

subroutine eval_node_set_op1_pdg (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_pdg) :: op
    en%op1_pdg => op
end subroutine eval_node_set_op1_pdg

subroutine eval_node_set_op1_sev (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_sev) :: op
    en%op1_sev => op
end subroutine eval_node_set_op1_sev

subroutine eval_node_set_op1_str (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(unary_str) :: op
    en%op1_str => op

```

```

end subroutine eval_node_set_op1_str

subroutine eval_node_set_op2_log (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_log) :: op
  en%op2_log => op
end subroutine eval_node_set_op2_log

subroutine eval_node_set_op2_int (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_int) :: op
  en%op2_int => op
end subroutine eval_node_set_op2_int

subroutine eval_node_set_op2_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_real) :: op
  en%op2_real => op
end subroutine eval_node_set_op2_real

  subroutine eval_node_set_op2_cmplx (en, op)
    type(eval_node_t), intent(inout) :: en
    procedure(binary_cmplx) :: op
    en%op2_cmplx => op
  end subroutine eval_node_set_op2_cmplx

subroutine eval_node_set_op2_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_pdg) :: op
  en%op2_pdg => op
end subroutine eval_node_set_op2_pdg

subroutine eval_node_set_op2_sev (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_sev) :: op
  en%op2_sev => op
end subroutine eval_node_set_op2_sev

subroutine eval_node_set_op2_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_str) :: op
  en%op2_str => op
end subroutine eval_node_set_op2_str

```

### 5.6.3 Specific operators

Our expression syntax contains all Fortran functions that make sense. These functions have to be provided in a form that they can be used in procedures pointers, and have the abstract interfaces above. For some intrinsic functions, we could use specific versions provided by Fortran directly. However, this has two drawbacks: (i) We should work with the values instead of the eval-nodes as argument, which complicates the interface; (ii) more importantly, the **default**

real type need not be equivalent to double precision. This would, at least, introduce system dependencies. Finally, for operators there are no specific versions.

Therefore, we write wrappers for all possible functions, at the expense of some overhead.

## Binary numerical functions

```

<Expressions: procedures>+≡
  integer function add_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival + en2%ival
  end function add_ii
  real(default) function add_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival + en2%rval
  end function add_ir
  complex(default) function add_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival + en2%cval
  end function add_ic
  real(default) function add_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval + en2%ival
  end function add_ri
  complex(default) function add_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval + en2%ival
  end function add_ci
  complex(default) function add_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval + en2%rval
  end function add_cr
  complex(default) function add_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval + en2%cval
  end function add_rc
  real(default) function add_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval + en2%rval
  end function add_rr
  complex(default) function add_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval + en2%cval
  end function add_cc

  integer function sub_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%ival
  end function sub_ii
  real(default) function sub_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%rval
  end function sub_ir
  real(default) function sub_ri (en1, en2) result (y)

```

```

        type(eval_node_t), intent(in) :: en1, en2
        y = en1%rval - en2%ival
    end function sub_ri
complex(default) function sub_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%cval
end function sub_ic
complex(default) function sub_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%ival
end function sub_ci
complex(default) function sub_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%rval
end function sub_cr
complex(default) function sub_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval - en2%cval
end function sub_rc
real(default) function sub_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval - en2%rval
end function sub_rr
complex(default) function sub_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%cval
end function sub_cc

integer function mul_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%ival
end function mul_ii
real(default) function mul_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%rval
end function mul_ir
real(default) function mul_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval * en2%ival
end function mul_ri
complex(default) function mul_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%cval
end function mul_ic
complex(default) function mul_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval * en2%ival
end function mul_ci
complex(default) function mul_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval * en2%cval
end function mul_rc
complex(default) function mul_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2

```

```

        y = en1%cval * en2%rval
    end function mul_cr
    real(default) function mul_rr (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%rval * en2%rval
    end function mul_rr
    complex(default) function mul_cc (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%cval * en2%cval
    end function mul_cc

    integer function div_ii (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (en2%ival == 0) then
            if (en1%ival >= 0) then
                call msg_warning ("division by zero: " // int2char (en1%ival) // &
                    " / 0 ; result set to 0")
            else
                call msg_warning ("division by zero: (" // int2char (en1%ival) // &
                    ") / 0 ; result set to 0")
            end if
            y = 0
            return
        end if
        y = en1%ival / en2%ival
    end function div_ii
    real(default) function div_ir (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%ival / en2%rval
    end function div_ir
    real(default) function div_ri (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%rval / en2%ival
    end function div_ri
    complex(default) function div_ic (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%ival / en2%cval
    end function div_ic
    complex(default) function div_ci (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%cval / en2%ival
    end function div_ci
    complex(default) function div_rc (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%rval / en2%cval
    end function div_rc
    complex(default) function div_cr (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%cval / en2%rval
    end function div_cr
    real(default) function div_rr (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%rval / en2%rval
    end function div_rr

```

```

complex(default) function div_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval / en2%cval
end function div_cc

integer function pow_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    integer :: a, b
    real(default) :: rres
    a = en1%ival
    b = en2%ival
    if ((a == 0) .and. (b < 0)) then
        call msg_warning ("division by zero: " // int2char (a) // &
            " ^ (" // int2char (b) // ") ; result set to 0")
        y = 0
        return
    end if
    rres = real(a, default) ** b
    y = rres
    if (real(y, default) /= rres) then
        if (b < 0) then
            call msg_warning ("result of all-integer operation " // &
                int2char (a) // " ^ (" // int2char (b) // &
                ") has been truncated to " // int2char (y), &
                (/ var_str ("Chances are that you want to use " // &
                    "reals instead of integers at this point.") //))
        else
            call msg_warning ("integer overflow in " // int2char (a) // &
                " ^ " // int2char (b) // " ; result is " // int2char (y), &
                (/ var_str ("Using reals instead of integers might help.") //))
        end if
    end if
end function pow_ii

real(default) function pow_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval ** en2%ival
end function pow_ri

complex(default) function pow_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval ** en2%ival
end function pow_ci

real(default) function pow_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival ** en2%rval
end function pow_ir

real(default) function pow_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval ** en2%rval
end function pow_rr

complex(default) function pow_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval ** en2%rval
end function pow_cr

complex(default) function pow_ic (en1, en2) result (y)

```

```

        type(eval_node_t), intent(in) :: en1, en2
        y = en1%ival ** en2%cval
    end function pow_ic
complex(default) function pow_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval ** en2%cval
end function pow_rc
complex(default) function pow_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval ** en2%cval
end function pow_cc

integer function max_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%ival, en2%ival)
end function max_ii
real(default) function max_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (real (en1%ival, default), en2%rval)
end function max_ir
real(default) function max_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%rval, real (en2%ival, default))
end function max_ri
real(default) function max_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%rval, en2%rval)
end function max_rr
integer function min_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%ival, en2%ival)
end function min_ii
real(default) function min_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (real (en1%ival, default), en2%rval)
end function min_ir
real(default) function min_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%rval, real (en2%ival, default))
end function min_ri
real(default) function min_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%rval, en2%rval)
end function min_rr

integer function mod_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%ival, en2%ival)
end function mod_ii
real(default) function mod_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (real (en1%ival, default), en2%rval)
end function mod_ir
real(default) function mod_ri (en1, en2) result (y)

```



```

        type(eval_node_t), intent(in) :: en1, en2
        y = mod (en1%rval, real (en2%ival, default))
    end function mod_ri
real(default) function mod_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%rval, en2%rval)
end function mod_rr
integer function modulo_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%ival, en2%ival)
end function modulo_ii
real(default) function modulo_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (real (en1%ival, default), en2%rval)
end function modulo_ir
real(default) function modulo_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, real (en2%ival, default))
end function modulo_ri
real(default) function modulo_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, en2%rval)
end function modulo_rr

```

## Unary numeric functions

*(Expressions: procedures)*+≡

```

real(default) function real_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
end function real_i
real(default) function real_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%cval
end function real_c
integer function int_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%rval
end function int_r
complex(default) function cmplx_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
end function cmplx_i
integer function int_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%cval
end function int_c
complex(default) function cmplx_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%rval
end function cmplx_r
integer function nint_r (en) result (y)
    type(eval_node_t), intent(in) :: en

```

```

    y = nint (en%rval)
end function nint_r
integer function floor_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = floor (en%rval)
end function floor_r
integer function ceiling_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = ceiling (en%rval)
end function ceiling_r

integer function neg_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = - en%ival
end function neg_i
real(default) function neg_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = - en%rval
end function neg_r
complex(default) function neg_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = - en%cval
end function neg_c
integer function abs_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = abs (en%ival)
end function abs_i
real(default) function abs_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = abs (en%rval)
end function abs_r
real(default) function abs_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = abs (en%cval)
end function abs_c
integer function sgn_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = sign (1, en%ival)
end function sgn_i
real(default) function sgn_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = sign (1._default, en%rval)
end function sgn_r

real(default) function sqrt_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = sqrt (en%rval)
end function sqrt_r
real(default) function exp_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = exp (en%rval)
end function exp_r
real(default) function log_r (en) result (y)
    type(eval_node_t), intent(in) :: en

```

```

        y = log (en%rval)
    end function log_r
    real(default) function log10_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = log10 (en%rval)
    end function log10_r

    complex(default) function sqrt_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = sqrt (en%cval)
    end function sqrt_c
    complex(default) function exp_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = exp (en%cval)
    end function exp_c
    complex(default) function log_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = log (en%cval)
    end function log_c

    real(default) function sin_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = sin (en%rval)
    end function sin_r
    real(default) function cos_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = cos (en%rval)
    end function cos_r
    real(default) function tan_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = tan (en%rval)
    end function tan_r
    real(default) function asin_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = asin (en%rval)
    end function asin_r
    real(default) function acos_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = acos (en%rval)
    end function acos_r
    real(default) function atan_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = atan (en%rval)
    end function atan_r

    complex(default) function sin_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = sin (en%cval)
    end function sin_c
    complex(default) function cos_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = cos (en%cval)
    end function cos_c

```

```

real(default) function sinh_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sinh (en%rval)
end function sinh_r
real(default) function cosh_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = cosh (en%rval)
end function cosh_r
real(default) function tanh_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = tanh (en%rval)
end function tanh_r
! real(default) function asinh_r (en) result (y)
!   type(eval_node_t), intent(in) :: en
!   y = asinh (en%rval)
! end function asinh_r
! real(default) function acosh_r (en) result (y)
!   type(eval_node_t), intent(in) :: en
!   y = acosh (en%rval)
! end function acosh_r
! real(default) function atanh_r (en) result (y)
!   type(eval_node_t), intent(in) :: en
!   y = atanh (en%rval)
! end function atanh_r

```

## Binary logical functions

Logical expressions:

*(Expressions: procedures)* +  $\equiv$

```

logical function ignore_first_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en2%lval
end function ignore_first_ll
logical function or_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%lval .or. en2%lval
end function or_ll
logical function and_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%lval .and. en2%lval
end function and_ll

```

Comparisons:

*(Expressions: procedures)* +  $\equiv$

```

logical function comp_lt_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival < en2%ival
end function comp_lt_ii
logical function comp_lt_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival < en2%rval
end function comp_lt_ir

```

```

logical function comp_lt_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval < en2%ival
end function comp_lt_ri
logical function comp_lt_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval < en2%rval
end function comp_lt_rr

logical function comp_gt_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival > en2%ival
end function comp_gt_ii
logical function comp_gt_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival > en2%rval
end function comp_gt_ir
logical function comp_gt_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval > en2%ival
end function comp_gt_ri
logical function comp_gt_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval > en2%rval
end function comp_gt_rr

logical function comp_le_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival <= en2%ival
end function comp_le_ii
logical function comp_le_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival <= en2%rval
end function comp_le_ir
logical function comp_le_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval <= en2%ival
end function comp_le_ri
logical function comp_le_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval <= en2%rval
end function comp_le_rr

logical function comp_ge_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival >= en2%ival
end function comp_ge_ii
logical function comp_ge_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival >= en2%rval
end function comp_ge_ir
logical function comp_ge_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval >= en2%ival

```

```

end function comp_ge_ri
logical function comp_ge_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval >= en2%rval
end function comp_ge_rr

logical function comp_eq_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival == en2%ival
end function comp_eq_ii
logical function comp_eq_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival == en2%rval
end function comp_eq_ir
logical function comp_eq_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval == en2%ival
end function comp_eq_ri
logical function comp_eq_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval == en2%rval
end function comp_eq_rr
logical function comp_eq_ss (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%sval == en2%sval
end function comp_eq_ss

logical function comp_ne_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival /= en2%ival
end function comp_ne_ii
logical function comp_ne_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival /= en2%rval
end function comp_ne_ir
logical function comp_ne_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval /= en2%ival
end function comp_ne_ri
logical function comp_ne_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval /= en2%rval
end function comp_ne_rr
logical function comp_ne_ss (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%sval /= en2%sval
end function comp_ne_ss

logical function comp_sim_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
        y = abs (en1%ival - en2%ival) <= en1%tolerance
    else
        y = en1%ival == en2%ival
    end if
end function comp_sim_ii

```

```

        end if
    end function comp_sim_ii
    logical function comp_sim_ri (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%rval - en2%ival) <= en1%tolerance
        else
            y = en1%rval == en2%ival
        end if
    end function comp_sim_ri
    logical function comp_sim_ir (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%ival - en2%rval) <= en1%tolerance
        else
            y = en1%ival == en2%rval
        end if
    end function comp_sim_ir
    logical function comp_sim_rr (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%rval - en2%rval) <= en1%tolerance
        else
            y = en1%rval == en2%rval
        end if
    end function comp_sim_rr
    logical function comp_nsim_ii (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%ival - en2%ival) > en1%tolerance
        else
            y = en1%ival /= en2%ival
        end if
    end function comp_nsim_ii
    logical function comp_nsim_ri (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%rval - en2%ival) > en1%tolerance
        else
            y = en1%rval /= en2%ival
        end if
    end function comp_nsim_ri
    logical function comp_nsim_ir (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%ival - en2%rval) > en1%tolerance
        else
            y = en1%ival /= en2%rval
        end if
    end function comp_nsim_ir
    logical function comp_nsim_rr (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        if (associated (en1%tolerance)) then
            y = abs (en1%rval - en2%rval) > en1%tolerance

```

```

    else
        y = en1%rval /= en2%rval
    end if
end function comp_nsim_rr

```

## Unary logical functions

*(Expressions: procedures)*+≡

```

logical function not_l (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = .not. en%lval
end function not_l

```

## Unary PDG-array functions

Make a PDG-array object from an integer.

*(Expressions: procedures)*+≡

```

subroutine pdg_i (pdg_array, en)
    type(pdg_array_t), intent(out) :: pdg_array
    type(eval_node_t), intent(in) :: en
    pdg_array = en%ival
end subroutine pdg_i

```

## Binary PDG-array functions

Concatenate two PDG-array objects.

*(Expressions: procedures)*+≡

```

subroutine concat_cc (pdg_array, en1, en2)
    type(pdg_array_t), intent(out) :: pdg_array
    type(eval_node_t), intent(in) :: en1, en2
    pdg_array = en1%aval // en2%aval
end subroutine concat_cc

```

## Unary particle-list functions

Combine all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test.

*(Expressions: procedures)*+≡

```

subroutine collect_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: n, i
    n = subevt_get_length (en1%pval)
    allocate (mask1 (n))
    if (present (en0)) then
        do i = 1, n

```



```

        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        mask1(i) = en0%lval
    end do
else
    mask1 = .true.
end if
call subevt_collect (subevt, en1%pval, mask1)
end subroutine collect_p

```

Select all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test.

*(Expressions: procedures)* +≡

```

subroutine select_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: n, i
    n = subevt_get_length (en1%pval)
    allocate (mask1 (n))
    if (present (en0)) then
        do i = 1, subevt_get_length (en1%pval)
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            call eval_node_evaluate (en0)
            mask1(i) = en0%lval
        end do
    else
        mask1 = .true.
    end if
    call subevt_select (subevt, en1%pval, mask1)
end subroutine select_p

```

Extract the particle with index given by `en0` from the argument list. Negative indices count from the end. If `en0` is absent, extract the first particle. The result is a list with a single entry, or no entries if the original list was empty or if the index is out of range.

This function has no counterpart with two arguments.

*(Expressions: procedures)* +≡

```

subroutine extract_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer :: index
    if (present (en0)) then
        call eval_node_evaluate (en0)
        select case (en0%result_type)
        case (V_INT); index = en0%ival
        case default
            call eval_node_write (en0)
            call msg_fatal (" Index parameter of 'extract' must be integer.")
        end select
    end if
end subroutine extract_p

```

```

        end select
    else
        index = 1
    end if
    call subevt_extract (subevt, en1%pval, index)
end subroutine extract_p

```

Sort the subevent according to the result of evaluating `en0`. If `en0` is absent, sort by default method (PDG code, particles before antiparticles).

*(Expressions: procedures)*+≡

```

subroutine sort_p (subevt, en1, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  integer, dimension(:), allocatable :: ival
  real(default), dimension(:), allocatable :: rval
  integer :: i, n
  n = subevt_get_length (en1%pval)
  if (present (en0)) then
    select case (en0%result_type)
    case (V_INT); allocate (ival (n))
    case (V_REAL); allocate (rval (n))
    end select
    do i = 1, n
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      call eval_node_evaluate (en0)
      select case (en0%result_type)
      case (V_INT); ival(i) = en0%ival
      case (V_REAL); rval(i) = en0%rval
      end select
    end do
    select case (en0%result_type)
    case (V_INT); call subevt_sort (subevt, en1%pval, ival)
    case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
    end select
  else
    call subevt_sort (subevt, en1%pval)
  end if
end subroutine sort_p

```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all elements of the subevent. `any` and `no` are analogous.

*(Expressions: procedures)*+≡

```

function all_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = subevt_get_length (en1%pval)
  lval = .true.
  do i = 1, n
    en0%index = i

```

```

        en0%prt1 = subevt_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        lval = en0%lval
        if (.not. lval) exit
    end do
end function all_p

function any_p (en1, en0) result (lval)
    logical :: lval
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout) :: en0
    integer :: i, n
    n = subevt_get_length (en1%pval)
    lval = .false.
    do i = 1, n
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        lval = en0%lval
        if (lval) exit
    end do
end function any_p

function no_p (en1, en0) result (lval)
    logical :: lval
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout) :: en0
    integer :: i, n
    n = subevt_get_length (en1%pval)
    lval = .true.
    do i = 1, n
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        lval = .not. en0%lval
        if (lval) exit
    end do
end function no_p

```

This is the interface to user-supplied observables. The node `en0` evaluates to a string that indicates the procedure name. We search for the procedure in the dynamic library and load it into the procedure pointer which is then called. `en1` is the subevent on which the external code operates. The external function returns a `c_int`, which we translate into a real value.

```

<Expressions: procedures>+≡
function user_obs_int_p (en0, prt1) result (ival)
    integer :: ival
    type(eval_node_t), intent(inout) :: en0
    type(prt_t), intent(in) :: prt1
    type(string_t) :: name
    procedure(user_obs_int_unary), pointer :: user_obs
    call eval_node_evaluate (en0)
    if (en0%value_is_known) then

```

```

        select case (en0%result_type)
        case (V_STR); name = en0%sval
        case default
            call msg_bug ("user_obs: procedure name must be a string")
            name = ""
        end select
        call c_f_procpointer (user_code_find_proc (name), user_obs)
        ival = user_obs (c_prt (prt1))
    else
        call eval_node_write_rec (en0)
        call msg_fatal ("User observable name is undefined")
    end if
end function user_obs_int_p

function user_obs_real_p (en0, prt1) result (rval)
    real(default) :: rval
    type(eval_node_t), intent(inout) :: en0
    type(prt_t), intent(in) :: prt1
    type(string_t) :: name
    procedure(user_obs_real_unary), pointer :: user_obs
    call eval_node_evaluate (en0)
    if (en0%value_is_known) then
        select case (en0%result_type)
        case (V_STR); name = en0%sval
        case default
            call msg_bug ("user_obs: procedure name must be a string")
            name = ""
        end select
        call c_f_procpointer (user_code_find_proc (name), user_obs)
        rval = user_obs (c_prt (prt1))
    else
        call eval_node_write_rec (en0)
        call msg_fatal ("User observable name is undefined")
    end if
end function user_obs_real_p

```

This is the interface to user-supplied cut code. The node `en0` evaluates to a string that indicates the procedure name.

*(Expressions: procedures)* +=

```

function user_cut_p (en1, en0) result (lval)
    logical :: lval
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout) :: en0
    type(string_t) :: name
    procedure(user_cut_fun), pointer :: user_cut
    call eval_node_evaluate (en0)
    select case (en0%result_type)
    case (V_STR); name = en0%sval
    case default
        call msg_bug ("user_cut: procedure name must be a string")
        name = ""
    end select
    call c_f_procpointer (user_code_find_proc (name), user_cut)

```

```

    lval = user_cut (c_prt (en1%pval), &
                    int (subevt_get_length (en1%pval), kind=c_int)) &
    /= 0
end function user_cut_p

```

The following function returns an integer value, namely the number of particles for which the condition is true. If there is no condition, it returns simply the length of the subevent.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface `unary_evi`.)

*(Expressions: procedures)*+≡

```

subroutine count_a (ival, en1, en0)
  integer, intent(out) :: ival
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  integer :: i, n, count
  n = subevt_get_length (en1%pval)
  if (present (en0)) then
    count = 0
    do i = 1, n
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      call eval_node_evaluate (en0)
      if (en0%lval) count = count + 1
    end do
    ival = count
  else
    ival = n
  end if
end subroutine count_a

```

This evaluates a user-defined event-shape observable for the current subevent.

*(Expressions: procedures)*+≡

```

subroutine user_event_shape_a (rval, en1, en0)
  real(default), intent(out) :: rval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  type(string_t) :: name
  procedure(user_event_shape_fun), pointer :: user_event_shape
  if (.not. present (en0)) call msg_bug &
    ("user_event_shape called without procedure name")
  call eval_node_evaluate (en0)
  select case (en0%result_type)
  case (V_STR); name = en0%sval
  case default
    call msg_bug ("user_event_shape: procedure name must be a string")
    name = ""
  end select
  call c_f_procpointer (user_code_find_proc (name), user_event_shape)
  rval = user_event_shape (c_prt (en1%pval), &
                          int (subevt_get_length (en1%pval), kind=c_int))
end subroutine user_event_shape_a

```

## Binary particle-list functions

This joins two subevents, stored in the evaluation nodes **en1** and **en2**. If **en0** is also present, it amounts to a logical test returning true or false for every pair of particles. A particle of the second list gets a mask entry only if it passes the test for all particles of the first list.

*(Expressions: procedures)*+≡

```
subroutine join_pp (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  logical, dimension(:), allocatable :: mask2
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  allocate (mask2 (n2))
  mask2 = .true.
  if (present (en0)) then
    do i = 1, n1
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        call eval_node_evaluate (en0)
        mask2(j) = mask2(j) .and. en0%lval
      end do
    end do
  end if
  call subevt_join (subevt, en1%pval, en2%pval, mask2)
end subroutine join_pp
```

Combine two subevents, i.e., make a list of composite particles built from all possible particle pairs from the two lists. If **en0** is present, create a mask which is true only for those pairs that pass the test.

*(Expressions: procedures)*+≡

```
subroutine combine_pp (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  logical, dimension(:,:), allocatable :: mask12
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  if (present (en0)) then
    allocate (mask12 (n1, n2))
    do i = 1, n1
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        call eval_node_evaluate (en0)
        mask12(i,j) = en0%lval
      end do
    end do
  end if
end subroutine combine_pp
```

```

        end do
        call subevt_combine (subevt, en1%pval, en2%pval, mask12)
    else
        call subevt_combine (subevt, en1%pval, en2%pval)
    end if
end subroutine combine_pp

```

Combine all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test w.r.t. all particles in the second argument. If `en0` is absent, the second argument is ignored.

*(Expressions: procedures)+≡*

```

subroutine collect_pp (subevt, en1, en2, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: i, j, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    allocate (mask1 (n1))
    mask1 = .true.
    if (present (en0)) then
        do i = 1, n1
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            do j = 1, n2
                en0%prt2 = subevt_get_prt (en2%pval, j)
                call eval_node_evaluate (en0)
                mask1(i) = mask1(i) .and. en0%lval
            end do
        end do
    end if
    call subevt_collect (subevt, en1%pval, mask1)
end subroutine collect_pp

```

Select all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test w.r.t. all particles in the second argument. If `en0` is absent, the second argument is ignored, and the first argument is transferred unchanged. (This case is not very useful, of course.)

*(Expressions: procedures)+≡*

```

subroutine select_pp (subevt, en1, en2, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: i, j, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    allocate (mask1 (n1))
    mask1 = .true.
    if (present (en0)) then
        do i = 1, n1
            en0%index = i

```

```

        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
            en0%prt2 = subevt_get_prt (en2%pval, j)
            call eval_node_evaluate (en0)
            mask1(i) = mask1(i) .and. en0%lval
        end do
    end do
end if
call subevt_select (subevt, en1%pval, mask1)
end subroutine select_pp

```

Sort the first subevent according to the result of evaluating `en0`. From the second subevent, only the first element is taken as reference. If `en0` is absent, we sort by default method (PDG code, particles before antiparticles).

*(Expressions: procedures)+≡*

```

subroutine sort_pp (subevt, en1, en2, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    integer, dimension(:), allocatable :: ival
    real(default), dimension(:), allocatable :: rval
    integer :: i, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    if (present (en0)) then
        select case (en0%result_type)
            case (V_INT); allocate (ival (n1))
            case (V_REAL); allocate (rval (n1))
        end select
        do i = 1, n1
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            en0%prt2 = subevt_get_prt (en2%pval, 1)
            call eval_node_evaluate (en0)
            select case (en0%result_type)
                case (V_INT); ival(i) = en0%ival
                case (V_REAL); rval(i) = en0%rval
            end select
        end do
        select case (en0%result_type)
            case (V_INT); call subevt_sort (subevt, en1%pval, ival)
            case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
        end select
    else
        call subevt_sort (subevt, en1%pval)
    end if
end subroutine sort_pp

```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all valid element pairs of both subevents. Invalid pairs (with common `src` entry) are ignored.

`any` and `no` are analogous.

*(Expressions: procedures)+≡*



```

function all_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  lval = .true.
  LOOP1: do i = 1, n1
    en0%index = i
    en0%prt1 = subevt_get_prt (en1%pval, i)
    do j = 1, n2
      en0%prt2 = subevt_get_prt (en2%pval, j)
      if (are_disjoint (en0%prt1, en0%prt2)) then
        call eval_node_evaluate (en0)
        lval = en0%lval
        if (.not. lval) exit LOOP1
      end if
    end do
  end do LOOP1
end function all_pp

function any_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  lval = .false.
  LOOP1: do i = 1, n1
    en0%index = i
    en0%prt1 = subevt_get_prt (en1%pval, i)
    do j = 1, n2
      en0%prt2 = subevt_get_prt (en2%pval, j)
      if (are_disjoint (en0%prt1, en0%prt2)) then
        call eval_node_evaluate (en0)
        lval = en0%lval
        if (lval) exit LOOP1
      end if
    end do
  end do LOOP1
end function any_pp

function no_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  lval = .true.
  LOOP1: do i = 1, n1
    en0%index = i

```

```

    en0%prt1 = subevt_get_prt (en1%pval, i)
    do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
            call eval_node_evaluate (en0)
            lval = .not. en0%lval
            if (lval) exit LOOP1
        end if
    end do
end do LOOP1
end function no_pp

```

This is the interface to user-supplied observables. The node `en0` evaluates to a string that indicates the procedure name. We search for the procedure in the dynamic library and load it into the procedure pointer which is then called. `en1` is the subevent on which the external code operates. The external function returns a `c_int`, which we translate into a real value.

*(Expressions: procedures)* +≡

```

function user_obs_int_pp (en0, prt1, prt2) result (ival)
    integer :: ival
    type(eval_node_t), intent(inout) :: en0
    type(prt_t), intent(in) :: prt1, prt2
    type(string_t) :: name
    procedure(user_obs_int_binary), pointer :: user_obs
    call eval_node_evaluate (en0)
    if (en0%value_is_known) then
        select case (en0%result_type)
            case (V_STR); name = en0%sval
            case default
                call msg_bug ("user_obs: procedure name must be a string")
                name = ""
        end select
        call c_f_procpointer (user_code_find_proc (name), user_obs)
        ival = user_obs (c_prt (prt1), c_prt (prt2))
    else
        call eval_node_write_rec (en0)
        call msg_fatal ("User observable name is undefined")
    end if
end function user_obs_int_pp

function user_obs_real_pp (en0, prt1, prt2) result (rval)
    real(default) :: rval
    type(eval_node_t), intent(inout) :: en0
    type(prt_t), intent(in) :: prt1, prt2
    type(string_t) :: name
    procedure(user_obs_real_binary), pointer :: user_obs
    call eval_node_evaluate (en0)
    if (en0%value_is_known) then
        select case (en0%result_type)
            case (V_STR); name = en0%sval
            case default
                call msg_bug ("user_obs: procedure name must be a string")
                name = ""
        end select
        call c_f_procpointer (user_code_find_proc (name), user_obs)
        rval = user_obs (c_prt (prt1), c_prt (prt2))
    else
        call eval_node_write_rec (en0)
        call msg_fatal ("User observable name is undefined")
    end if
end function user_obs_real_pp

```

```

        end select
        call c_f_procpointer (user_code_find_proc (name), user_obs)
        rval = user_obs (c_prt (prt1), c_prt (prt2))
    else
        call eval_node_write_rec (en0)
        call msg_fatal ("User observable name is undefined")
    end if
end function user_obs_real_pp

```

The following function returns an integer value, namely the number of valid particle-pairs from both lists for which the condition is true. Invalid pairs (with common `src` entry) are ignored. If there is no condition, it returns the number of valid particle pairs.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface `binary_num`.)

*(Expressions: procedures)*+≡

```

subroutine count_pp (ival, en1, en2, en0)
    integer, intent(out) :: ival
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    integer :: i, j, n1, n2, count
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    if (present (en0)) then
        count = 0
        do i = 1, n1
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            do j = 1, n2
                en0%prt2 = subevt_get_prt (en2%pval, j)
                if (are_disjoint (en0%prt1, en0%prt2)) then
                    call eval_node_evaluate (en0)
                    if (en0%lval) count = count + 1
                end if
            end do
        end do
    else
        count = 0
        do i = 1, n1
            do j = 1, n2
                if (are_disjoint (subevt_get_prt (en1%pval, i), &
                                     subevt_get_prt (en2%pval, j))) then
                    count = count + 1
                end if
            end do
        end do
    end if
    ival = count
end subroutine count_pp

```

This function makes up a subevent from the second argument which consists only of particles which match the PDG code array (first argument).

```

<Expressions: procedures>+≡
subroutine select_pdg_ca (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  if (present (en0)) then
    call subevt_select_pdg_code (subevt, en1%aval, en2%pval, en0%ival)
  else
    call subevt_select_pdg_code (subevt, en1%aval, en2%pval)
  end if
end subroutine select_pdg_ca

```

## Binary string functions

Currently, the only string operation is concatenation.

```

<Expressions: procedures>+≡
subroutine concat_ss (string, en1, en2)
  type(string_t), intent(out) :: string
  type(eval_node_t), intent(in) :: en1, en2
  string = en1%sval // en2%sval
end subroutine concat_ss

```

## 5.6.4 Compiling the parse tree

The evaluation tree is built recursively by following a parse tree. Debug option:

```

<Expressions: variables>≡
logical, parameter :: debug = .false.

```

Evaluate an expression. The requested type is given as an optional argument; default is numeric (integer or real).

```

<Expressions: procedures>+≡
recursive subroutine eval_node_compile_genexpr &
  (en, pn, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: result_type
  if (debug) then
    print *, "read genexpr"; call parse_node_write (pn)
  end if
  if (present (result_type)) then
    select case (result_type)
    case (V_INT, V_REAL, V_CMPLX)
      call eval_node_compile_expr (en, pn, var_list)
    case (V_LOG)
      call eval_node_compile_lexpr (en, pn, var_list)
    case (V_SEV)
      call eval_node_compile_pexpr (en, pn, var_list)
    case (V_PDG)
      call eval_node_compile_cexpr (en, pn, var_list)

```

```

        case (V_STR)
            call eval_node_compile_sexpr (en, pn, var_list)
        end select
    else
        call eval_node_compile_expr (en, pn, var_list)
    end if
    if (debug) then
        call eval_node_write (en)
        print *, "done genexpr"
    end if
end subroutine eval_node_compile_genexpr

```

## Numeric expressions

This procedure compiles a numerical expression. This is a single term or a sum or difference of terms. We have to account for all combinations of integer and real arguments. If both are constant, we immediately do the calculation and allocate a constant node.

*(Expressions: procedures)* +≡

```

recursive subroutine eval_node_compile_expr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_term, pn_addition, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2, t
    if (debug) then
        print *, "read expr"; call parse_node_write (pn)
    end if
    pn_term => parse_node_get_sub_ptr (pn)
    select case (char (parse_node_get_rule_key (pn_term)))
    case ("term")
        call eval_node_compile_term (en, pn_term, var_list)
        pn_addition => parse_node_get_next_ptr (pn_term, tag="addition")
    case ("addition")
        en => null ()
        pn_addition => pn_term
    case default
        call parse_node_mismatch ("term|addition", pn)
    end select
    do while (associated (pn_addition))
        pn_op => parse_node_get_sub_ptr (pn_addition)
        pn_arg => parse_node_get_next_ptr (pn_op, tag="term")
        call eval_node_compile_term (en2, pn_arg, var_list)
        t2 = en2%result_type
        if (associated (en)) then
            en1 => en
            t1 = en1%result_type
        else
            allocate (en1)
            select case (t2)

```

```

case (V_INT); call eval_node_init_int (en1, 0)
case (V_REAL); call eval_node_init_real (en1, 0._default)
case (V_CMPLX); call eval_node_init_cmplx (en1, cmplx &
                                (0._default, 0._default, kind=default))
end select
t1 = t2
end if
t = numeric_result_type (t1, t2)
allocate (en)
key = parse_node_get_key (pn_op)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
select case (char (key))
case ("+")
select case (t1)
case (V_INT)
select case (t2)
case (V_INT); call eval_node_init_int (en, add_ii (en1, en2))
case (V_REAL); call eval_node_init_real (en, add_ir (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, add_ic (en1, en2))
end select
case (V_REAL)
select case (t2)
case (V_INT); call eval_node_init_real (en, add_ri (en1, en2))
case (V_REAL); call eval_node_init_real (en, add_rr (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, add_rc (en1, en2))
end select
case (V_CMPLX)
select case (t2)
case (V_INT); call eval_node_init_cmplx (en, add_ci (en1, en2))
case (V_REAL); call eval_node_init_cmplx (en, add_cr (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, add_cc (en1, en2))
end select
end select
case ("-")
select case (t1)
case (V_INT)
select case (t2)
case (V_INT); call eval_node_init_int (en, sub_ii (en1, en2))
case (V_REAL); call eval_node_init_real (en, sub_ir (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, sub_ic (en1, en2))
end select
case (V_REAL)
select case (t2)
case (V_INT); call eval_node_init_real (en, sub_ri (en1, en2))
case (V_REAL); call eval_node_init_real (en, sub_rr (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, sub_rc (en1, en2))
end select
case (V_CMPLX)
select case (t2)
case (V_INT); call eval_node_init_cmplx (en, sub_ci (en1, en2))
case (V_REAL); call eval_node_init_cmplx (en, sub_cr (en1, en2))
case (V_CMPLX); call eval_node_init_cmplx (en, sub_cc (en1, en2))
end select
end select
end select

```

```

end select
call eval_node_final_rec (en1)
call eval_node_final_rec (en2)
deallocate (en1, en2)
else
call eval_node_init_branch (en, key, t, en1, en2)
select case (char (key))
case ("+")
select case (t1)
case (V_INT)
select case (t2)
case (V_INT); call eval_node_set_op2_int (en, add_ii)
case (V_REAL); call eval_node_set_op2_real (en, add_ir)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, add_ic)
end select
case (V_REAL)
select case (t2)
case (V_INT); call eval_node_set_op2_real (en, add_ri)
case (V_REAL); call eval_node_set_op2_real (en, add_rr)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, add_rc)
end select
case (V_CMPLX)
select case (t2)
case (V_INT); call eval_node_set_op2_cmplx (en, add_ci)
case (V_REAL); call eval_node_set_op2_cmplx (en, add_cr)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, add_cc)
end select
end select
case ("-")
select case (t1)
case (V_INT)
select case (t2)
case (V_INT); call eval_node_set_op2_int (en, sub_ii)
case (V_REAL); call eval_node_set_op2_real (en, sub_ir)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, sub_ic)
end select
case (V_REAL)
select case (t2)
case (V_INT); call eval_node_set_op2_real (en, sub_ri)
case (V_REAL); call eval_node_set_op2_real (en, sub_rr)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, sub_rc)
end select
case (V_CMPLX)
select case (t2)
case (V_INT); call eval_node_set_op2_cmplx (en, sub_ci)
case (V_REAL); call eval_node_set_op2_cmplx (en, sub_cr)
case (V_CMPLX); call eval_node_set_op2_cmplx (en, sub_cc)
end select
end select
end select
end if
pn_addition => parse_node_get_next_ptr (pn_addition)
end do
if (debug) then

```

```

        call eval_node_write (en)
        print *, "done expr"
    end if
end subroutine eval_node_compile_expr

```

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_term (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_factor, pn_multiplication, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2, t
    if (debug) then
        print *, "read term"; call parse_node_write (pn)
    end if
    pn_factor => parse_node_get_sub_ptr (pn, tag="factor")
    call eval_node_compile_factor (en, pn_factor, var_list)
    pn_multiplication => &
        parse_node_get_next_ptr (pn_factor, tag="multiplication")
    do while (associated (pn_multiplication))
        pn_op => parse_node_get_sub_ptr (pn_multiplication)
        pn_arg => parse_node_get_next_ptr (pn_op, tag="factor")
        en1 => en
        call eval_node_compile_factor (en2, pn_arg, var_list)
        t1 = en1%result_type
        t2 = en2%result_type
        t = numeric_result_type (t1, t2)
        allocate (en)
        key = parse_node_get_key (pn_op)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            select case (char (key))
            case ("*")
                select case (t1)
                case (V_INT)
                    select case (t2)
                    case (V_INT); call eval_node_init_int (en, mul_ii (en1, en2))
                    case (V_REAL); call eval_node_init_real (en, mul_ir (en1, en2))
                    case (V_CMPLX); call eval_node_init_cmplx (en, mul_ic (en1, en2))
                    end select
                case (V_REAL)
                    select case (t2)
                    case (V_INT); call eval_node_init_real (en, mul_ri (en1, en2))
                    case (V_REAL); call eval_node_init_real (en, mul_rr (en1, en2))
                    case (V_CMPLX); call eval_node_init_cmplx (en, mul_rc (en1, en2))
                    end select
                case (V_CMPLX)
                    select case (t2)
                    case (V_INT); call eval_node_init_cmplx (en, mul_ci (en1, en2))
                    case (V_REAL); call eval_node_init_cmplx (en, mul_cr (en1, en2))
                    case (V_CMPLX); call eval_node_init_cmplx (en, mul_cc (en1, en2))
                    end select
            end select
        end select
    end while
end subroutine eval_node_compile_term

```



```

case ("/")
  select case (t1)
    case (V_INT)
      select case (t2)
        case (V_INT); call eval_node_init_int (en, div_ii (en1, en2))
        case (V_REAL); call eval_node_init_real (en, div_ir (en1, en2))
        case (V_CMPLX); call eval_node_init_real (en, div_ir (en1, en2))
      end select
    case (V_REAL)
      select case (t2)
        case (V_INT); call eval_node_init_real (en, div_ri (en1, en2))
        case (V_REAL); call eval_node_init_real (en, div_rr (en1, en2))
        case (V_CMPLX); call eval_node_init_cmplx (en, div_rc (en1, en2))
      end select
    case (V_CMPLX)
      select case (t2)
        case (V_INT); call eval_node_init_cmplx (en, div_ci (en1, en2))
        case (V_REAL); call eval_node_init_cmplx (en, div_cr (en1, en2))
        case (V_CMPLX); call eval_node_init_cmplx (en, div_cc (en1, en2))
      end select
  end select
call eval_node_final_rec (en1)
call eval_node_final_rec (en2)
deallocate (en1, en2)
else
  call eval_node_init_branch (en, key, t, en1, en2)
  select case (char (key))
    case ("*")
      select case (t1)
        case (V_INT)
          select case (t2)
            case (V_INT); call eval_node_set_op2_int (en, mul_ii)
            case (V_REAL); call eval_node_set_op2_real (en, mul_ir)
            case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_ic)
          end select
        case (V_REAL)
          select case (t2)
            case (V_INT); call eval_node_set_op2_real (en, mul_ri)
            case (V_REAL); call eval_node_set_op2_real (en, mul_rr)
            case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_rc)
          end select
        case (V_CMPLX)
          select case (t2)
            case (V_INT); call eval_node_set_op2_cmplx (en, mul_ci)
            case (V_REAL); call eval_node_set_op2_cmplx (en, mul_cr)
            case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_cc)
          end select
      end select
    case ("/")
      select case (t1)
        case (V_INT)
          select case (t2)
            case (V_INT); call eval_node_set_op2_int (en, div_ii)

```

```

        case (V_REAL); call eval_node_set_op2_real (en, div_ir)
        case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_ic)
    end select
    case (V_REAL)
        select case (t2)
            case (V_INT); call eval_node_set_op2_real (en, div_ri)
            case (V_REAL); call eval_node_set_op2_real (en, div_rr)
            case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_rc)
        end select
    case (V_CMPLX)
        select case (t2)
            case (V_INT); call eval_node_set_op2_cmplx (en, div_ci)
            case (V_REAL); call eval_node_set_op2_cmplx (en, div_cr)
            case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_cc)
        end select
    end select
end select
end if
pn_multiplication => parse_node_get_next_ptr (pn_multiplication)
end do
if (debug) then
    call eval_node_write (en)
    print *, "done term"
end if
end subroutine eval_node_compile_term

```

*(Expressions: procedures)* +=

```

recursive subroutine eval_node_compile_factor (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_value, pn_exponentiation, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2, t
    if (debug) then
        print *, "read factor"; call parse_node_write (pn)
    end if
    pn_value => parse_node_get_sub_ptr (pn)
    call eval_node_compile_signed_value (en, pn_value, var_list)
    pn_exponentiation => &
        parse_node_get_next_ptr (pn_value, tag="exponentiation")
    if (associated (pn_exponentiation)) then
        pn_op => parse_node_get_sub_ptr (pn_exponentiation)
        pn_arg => parse_node_get_next_ptr (pn_op)
        en1 => en
        call eval_node_compile_signed_value (en2, pn_arg, var_list)
        t1 = en1%result_type
        t2 = en2%result_type
        t = numeric_result_type (t1, t2)
        allocate (en)
        key = parse_node_get_key (pn_op)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            select case (t1)

```

```

case (V_INT)
  select case (t2)
    case (V_INT); call eval_node_init_int (en, pow_ii (en1, en2))
    case (V_REAL); call eval_node_init_real (en, pow_ir (en1, en2))
    case (V_CMPLX); call eval_node_init_cmplx (en, pow_ic (en1, en2))
  end select
case (V_REAL)
  select case (t2)
    case (V_INT); call eval_node_init_real (en, pow_ri (en1, en2))
    case (V_REAL); call eval_node_init_real (en, pow_rr (en1, en2))
    case (V_CMPLX); call eval_node_init_cmplx (en, pow_rc (en1, en2))
  end select
case (V_CMPLX)
  select case (t2)
    case (V_INT); call eval_node_init_cmplx (en, pow_ci (en1, en2))
    case (V_REAL); call eval_node_init_cmplx (en, pow_cr (en1, en2))
    case (V_CMPLX); call eval_node_init_cmplx (en, pow_cc (en1, en2))
  end select
end select
call eval_node_final_rec (en1)
call eval_node_final_rec (en2)
deallocate (en1, en2)
else
  call eval_node_init_branch (en, key, t, en1, en2)
  select case (t1)
    case (V_INT)
      select case (t2)
        case (V_INT); call eval_node_set_op2_int (en, pow_ii)
        case (V_REAL,V_CMPLX); call eval_type_error (pn, "exponentiation", t1)
      end select
    case (V_REAL)
      select case (t2)
        case (V_INT); call eval_node_set_op2_real (en, pow_ri)
        case (V_REAL); call eval_node_set_op2_real (en, pow_rr)
        case (V_CMPLX); call eval_type_error (pn, "exponentiation", t1)
      end select
    case (V_CMPLX)
      select case (t2)
        case (V_INT); call eval_node_set_op2_cmplx (en, pow_ci)
        case (V_REAL); call eval_node_set_op2_cmplx (en, pow_cr)
        case (V_CMPLX); call eval_node_set_op2_cmplx (en, pow_cc)
      end select
  end select
end if
end if
if (debug) then
  call eval_node_write (en)
  print *, "done factor"
end if
end subroutine eval_node_compile_factor

```

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_signed_value (en, pn, var_list)
  type(eval_node_t), pointer :: en

```

```

type(parse_node_t), intent(in) :: pn
type(var_list_t), intent(in), target :: var_list
type(parse_node_t), pointer :: pn_arg
type(eval_node_t), pointer :: en1
integer :: t
if (debug) then
    print *, "read signed value"; call parse_node_write (pn)
end if
select case (char (parse_node_get_rule_key (pn)))
case ("signed_value")
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    call eval_node_compile_value (en1, pn_arg, var_list)
    t = en1%result_type
    allocate (en)
    if (en1%type == EN_CONSTANT) then
        select case (t)
        case (V_INT); call eval_node_init_int (en, neg_i (en1))
        case (V_REAL); call eval_node_init_real (en, neg_r (en1))
        case (V_CMPLX); call eval_node_init_cmplx (en, neg_c (en1))
        end select
        call eval_node_final_rec (en1)
        deallocate (en1)
    else
        call eval_node_init_branch (en, var_str ("-"), t, en1)
        select case (t)
        case (V_INT); call eval_node_set_op1_int (en, neg_i)
        case (V_REAL); call eval_node_set_op1_real (en, neg_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, neg_c)
        end select
    end if
case default
    call eval_node_compile_value (en, pn, var_list)
end select
if (debug) then
    call eval_node_write (en)
    print *, "done signed value"
end if
end subroutine eval_node_compile_signed_value

```

Integer, real and complex values have an optional unit. The unit is extracted and applied immediately. An integer with unit evaluates to a real constant.

*(Expressions: procedures)* +≡

```

recursive subroutine eval_node_compile_value (en, pn, var_list)
type(eval_node_t), pointer :: en
type(parse_node_t), intent(in) :: pn
type(var_list_t), intent(in), target :: var_list
if (debug) then
    print *, "read value"; call parse_node_write (pn)
end if
select case (char (parse_node_get_rule_key (pn)))
case ("integer_value", "real_value", "complex_value")
    call eval_node_compile_numeric_value (en, pn)
case ("pi")

```

```

        call eval_node_compile_constant (en, pn)
case ("I")
    call eval_node_compile_constant (en, pn)
case ("variable")
    call eval_node_compile_variable (en, pn, var_list)
case ("result")
    call eval_node_compile_result (en, pn, var_list)
case ("user_observable")
    call eval_node_compile_user_observable (en, pn, var_list)
case ("expr")
    call eval_node_compile_expr (en, pn, var_list)
case ("block_expr")
    call eval_node_compile_block_expr (en, pn, var_list)
case ("conditional_expr")
    call eval_node_compile_conditional (en, pn, var_list)
case ("unary_function")
    call eval_node_compile_unary_function (en, pn, var_list)
case ("binary_function")
    call eval_node_compile_binary_function (en, pn, var_list)
case ("eval_fun")
    call eval_node_compile_eval_function (en, pn, var_list)
case ("count_fun", "user_event_fun")
    call eval_node_compile_numeric_function (en, pn, var_list)
case default
    call parse_node_mismatch &
        ("integer|real|complex|constant|variable|" // &
         "expr|block_expr|conditional_expr|" // &
         "unary_function|binary_function|numeric_pexpr", pn)
end select
if (debug) then
    call eval_node_write (en)
    print *, "done value"
end if
end subroutine eval_node_compile_value

```

Real, complex and integer values are numeric literals with an optional unit attached. In case of an integer, the unit actually makes it a real value in disguise. The signed version of real values is not possible in generic expressions; it is a special case for numeric constants in model files (see below). We do not introduce signed versions of complex values.

*(Expressions: procedures)* +≡

```

subroutine eval_node_compile_numeric_value (en, pn)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in), target :: pn
    type(parse_node_t), pointer :: pn_val, pn_unit
    allocate (en)
    pn_val => parse_node_get_sub_ptr (pn)
    pn_unit => parse_node_get_next_ptr (pn_val)
    select case (char (parse_node_get_rule_key (pn)))
case ("integer_value")
    if (associated (pn_unit)) then
        call eval_node_init_real (en, &
            parse_node_get_integer (pn_val) * parse_node_get_unit (pn_unit))

```

```

        else
            call eval_node_init_int (en, parse_node_get_integer (pn_val))
        end if
    case ("real_value")
        if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
        else
            call eval_node_init_real (en, parse_node_get_real (pn_val))
        end if
    case ("complex_value")
        if (associated (pn_unit)) then
            call eval_node_init_cmplx (en, &
                parse_node_get_cmplx (pn_val) * parse_node_get_unit (pn_unit))
        else
            call eval_node_init_cmplx (en, parse_node_get_cmplx (pn_val))
        end if
    case ("neg_real_value")
        pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
        pn_unit => parse_node_get_next_ptr (pn_val)
        if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                - parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
        else
            call eval_node_init_real (en, - parse_node_get_real (pn_val))
        end if
    case ("pos_real_value")
        pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
        pn_unit => parse_node_get_next_ptr (pn_val)
        if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
        else
            call eval_node_init_real (en, parse_node_get_real (pn_val))
        end if
    case default
        call parse_node_mismatch &
            ("integer_value|real_value|complex_value|neg_real_value|pos_real_value", pn)
    end select
end subroutine eval_node_compile_numeric_value

```

These are the units, predefined and hardcoded. The default energy unit is GeV, the default angular unit is radians. We include units for observables of dimension energy squared. Luminosities are normalized in inverse femtobarns.

*(Expressions: procedures)+≡*

```

function parse_node_get_unit (pn) result (factor)
    real(default) :: factor
    real(default) :: unit
    type(parse_node_t), intent(in) :: pn
    type(parse_node_t), pointer :: pn_unit, pn_unit_power
    type(parse_node_t), pointer :: pn_frac, pn_num, pn_int, pn_div, pn_den
    integer :: num, den
    pn_unit => parse_node_get_sub_ptr (pn)

```

```

select case (char (parse_node_get_key (pn_unit)))
case ("TeV"); unit = 1.e3_default
case ("GeV"); unit = 1
case ("MeV"); unit = 1.e-3_default
case ("keV"); unit = 1.e-6_default
case ("eV"); unit = 1.e-9_default
case ("meV"); unit = 1.e-12_default
case ("nbarn"); unit = 1.e6_default
case ("pbarn"); unit = 1.e3_default
case ("fbarn"); unit = 1
case ("abarn"); unit = 1.e-3_default
case ("rad"); unit = 1
case ("mrad"); unit = 1.e-3_default
case ("degree"); unit = degree
case ("%"); unit = 1.e-2_default
case default
    call msg_bug (" Unit ' ' // &
        char (parse_node_get_key (pn)) // " ' is undefined.")
end select
pn_unit_power => parse_node_get_next_ptr (pn_unit)
if (associated (pn_unit_power)) then
    pn_frac => parse_node_get_sub_ptr (pn_unit_power, 2)
    pn_num => parse_node_get_sub_ptr (pn_frac)
    select case (char (parse_node_get_rule_key (pn_num)))
    case ("neg_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = - parse_node_get_integer (pn_int)
    case ("pos_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = parse_node_get_integer (pn_int)
    case ("integer_literal")
        num = parse_node_get_integer (pn_num)
    case default
        call parse_node_mismatch ("neg_int|pos_int|integer_literal", pn_num)
    end select
    pn_div => parse_node_get_next_ptr (pn_num)
    if (associated (pn_div)) then
        pn_den => parse_node_get_sub_ptr (pn_div, 2)
        den = parse_node_get_integer (pn_den)
    else
        den = 1
    end if
else
    num = 1
    den = 1
end if
factor = unit ** (real (num, default) / den)
end function parse_node_get_unit

```

There are only two predefined constants, but more can be added easily.

*<Expressions: procedures>+≡*

```

subroutine eval_node_compile_constant (en, pn)
type(eval_node_t), pointer :: en
type(parse_node_t), intent(in) :: pn

```

```

if (debug) then
  print *, "read constant"; call parse_node_write (pn)
end if
allocate (en)
select case (char (parse_node_get_key (pn)))
case ("pi");      call eval_node_init_real (en, pi)
case ("I");       call eval_node_init_cmplx (en, imago)
case default
  call parse_node_mismatch ("pi or I", pn)
end select
if (debug) then
  call eval_node_write (en)
  print *, "done constant"
end if
end subroutine eval_node_compile_constant

```

Compile a variable, with or without a specified type. Take the list of variables, look for the name and make a node with a pointer to the value. If no type is provided, the variable is numeric, and the stored value determines whether it is real or integer.

We explicitly demand that the variable is defined, so we do not accidentally point to variables that are declared only later in the script but have come into existence in a previous compilation pass.

Variables may actually be anonymous, these are expressions in disguise. In that case, the expression replaces the variable name in the parse tree, and we allocate an ordinary expression node in the eval tree.

Variables of type V\_PDG (pdg-code array) are not treated here. They are handled by eval\_node\_compile\_cvariable.

*(Expressions: procedures)+≡*

```

recursive subroutine eval_node_compile_variable (en, pn, var_list, var_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: var_type
  type(parse_node_t), pointer :: pn_name
  type(string_t) :: var_name
  type(var_entry_t), pointer :: var
  logical, target, save :: no_lval
  real(default), target, save :: no_rval
  type(subvt_t), target, save :: no_pval
  type(string_t), target, save :: no_sval
  logical, target, save :: unknown = .false.
  if (debug) then
    print *, "read variable"; call parse_node_write (pn)
  end if
  if (present (var_type)) then
    select case (var_type)
    case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
          V_OBS2_INT, V_CMPLX)
      pn_name => pn
    case default
      pn_name => parse_node_get_sub_ptr (pn, 2)
    end select
  end if
end subroutine eval_node_compile_variable

```



```

else
    pn_name => pn
end if
select case (char (parse_node_get_rule_key (pn_name)))
case ("expr")
    call eval_node_compile_expr (en, pn_name, var_list)
case ("lexpr")
    call eval_node_compile_lexpr (en, pn_name, var_list)
case ("sexpr")
    call eval_node_compile_sexpr (en, pn_name, var_list)
case ("pexpr")
    call eval_node_compile_pexpr (en, pn_name, var_list)
case ("variable")
    var_name = parse_node_get_string (pn_name)
    if (present (var_type)) then
        select case (var_type)
        case (V_LOG); var_name = "?" // var_name
        case (V_SEV); var_name = "@" // var_name
        case (V_STR); var_name = "$" // var_name ! $ sign
        end select
    end if
    var => var_list_get_var_ptr &
        (var_list, var_name, var_type, defined=.true.)
    allocate (en)
    if (associated (var)) then
        select case (var_entry_get_type (var))
        case (V_LOG)
            call eval_node_init_log_ptr &
                (en, var_entry_get_name (var), var_entry_get_lval_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_INT)
            call eval_node_init_int_ptr &
                (en, var_entry_get_name (var), var_entry_get_ival_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_REAL)
            call eval_node_init_real_ptr &
                (en, var_entry_get_name (var), var_entry_get_rval_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_CMPLX)
            call eval_node_init_cmplx_ptr &
                (en, var_entry_get_name (var), var_entry_get_cval_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_SEV)
            call eval_node_init_subevt_ptr &
                (en, var_entry_get_name (var), var_entry_get_pval_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_STR)
            call eval_node_init_string_ptr &
                (en, var_entry_get_name (var), var_entry_get_sval_ptr (var), &
                var_entry_get_known_ptr (var))
        case (V_OBS1_INT, V_OBS2_INT, V_OBS1_REAL, V_OBS2_REAL)
            call eval_node_init_obs (en, var)
        case default
            call parse_node_write (pn)

```

```

        call msg_fatal ("Variable of this type " // &
            "is not allowed in the present context")
    if (present (var_type)) then
        select case (var_type)
        case (V_LOG)
            call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
        case (V_SEV)
            call eval_node_init_subevt_ptr &
                (en, var_name, no_pval, unknown)
        case (V_STR)
            call eval_node_init_string_ptr &
                (en, var_name, no_sval, unknown)
        end select
    else
        call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
    end if
end select
else
    call parse_node_write (pn)
    call msg_error ("This variable is undefined at this point")
    if (present (var_type)) then
        select case (var_type)
        case (V_LOG)
            call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
        case (V_SEV)
            call eval_node_init_subevt_ptr &
                (en, var_name, no_pval, unknown)
        case (V_STR)
            call eval_node_init_string_ptr (en, var_name, no_sval, unknown)
        end select
    else
        call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
    end if
end if
end select
if (debug) then
    call eval_node_write (en)
    print *, "done variable"
end if
end subroutine eval_node_compile_variable

```

In a given context, a variable has to have a certain type.

*(Expressions: procedures)* +≡

```

subroutine check_var_type (pn, ok, type_actual, type_requested)
    type(parse_node_t), intent(in) :: pn
    logical, intent(out) :: ok
    integer, intent(in) :: type_actual
    integer, intent(in), optional :: type_requested
    if (present (type_requested)) then
        select case (type_requested)
        case (V_LOG)
            select case (type_actual)
            case (V_LOG)
                case default

```

```

        call parse_node_write (pn)
        call msg_fatal ("Variable type is invalid (should be logical)")
        ok = .false.
    end select
case (V_SEV)
    select case (type_actual)
    case (V_SEV)
    case default
        call parse_node_write (pn)
        call msg_fatal &
            ("Variable type is invalid (should be particle set)")
        ok = .false.
    end select
case (V_PDG)
    select case (type_actual)
    case (V_PDG)
    case default
        call parse_node_write (pn)
        call msg_fatal &
            ("Variable type is invalid (should be PDG array)")
        ok = .false.
    end select
case (V_STR)
    select case (type_actual)
    case (V_STR)
    case default
        call parse_node_write (pn)
        call msg_fatal &
            ("Variable type is invalid (should be string)")
        ok = .false.
    end select
case default
    call parse_node_write (pn)
    call msg_bug ("Variable type is unknown")
end select
else
    select case (type_actual)
    case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
        V_OBS2_INT, V_CMPLX)
    case default
        call parse_node_write (pn)
        call msg_fatal ("Variable type is invalid (should be numeric)")
        ok = .false.
    end select
end if
ok = .true.
end subroutine check_var_type

```

Retrieve the result of an integration. If the requested process has been integrated, the results are available as special variables. (The variables cannot be accessed in the usual way since they contain brackets in their names.)

Since this compilation step may occur before the processes have been loaded, we have to initialize the required variables before they are used.

*(Expressions: procedures)*+≡

```

subroutine eval_node_compile_result (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_prc_id
  type(string_t) :: key, prc_id, var_name
  type(var_entry_t), pointer :: var
  if (debug) then
    print *, "read result"; call parse_node_write (pn)
  end if
  pn_key => parse_node_get_sub_ptr (pn)
  pn_prc_id => parse_node_get_next_ptr (pn_key)
  key = parse_node_get_key (pn_key)
  prc_id = parse_node_get_string (pn_prc_id)
  var_name = key // "(" // prc_id // ")"
  var => var_list_get_var_ptr (var_list, var_name)
  if (associated (var)) then
    allocate (en)
    select case (char(key))
      case ("num_id", "n_calls")
        call eval_node_init_int_ptr &
          (en, var_name, var_entry_get_ival_ptr (var), &
            var_entry_get_known_ptr (var))
      case ("integral", "error", "accuracy", "chi2", "efficiency")
        call eval_node_init_real_ptr &
          (en, var_name, var_entry_get_rval_ptr (var), &
            var_entry_get_known_ptr (var))
    end select
  else
    call msg_fatal ("Result variable '" // char (var_name) &
      // "' is undefined (call 'integrate' before use)")
  end if
  if (debug) then
    call eval_node_write (en)
    print *, "done result"
  end if
end subroutine eval_node_compile_result

```

This user observable behaves like a variable. We link the node to the generic user-observable entry in the variable list. The syntax element has an argument which provides the name of the user variable, this is stored as an eval-node alongside with the variable. When the variable value is used, the user-supplied external function is called and provides the (real) result value.

*(Expressions: procedures)*+≡

```

subroutine eval_node_compile_user_observable (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_arg, pn_obs
  type(eval_node_t), pointer :: en0
  integer :: res_type
  type(string_t) :: var_name

```

```

type(var_entry_t), pointer :: var
if (debug) then
    print *, "read user observable"; call parse_node_write (pn)
end if
pn_key => parse_node_get_sub_ptr (pn)
select case (char (parse_node_get_key (pn_key)))
case ("user_obs")
    res_type = V_REAL
case default
    call parse_node_write (pn_key)
    call msg_bug ("user_observable: wrong keyword")
end select
pn_arg => parse_node_get_next_ptr (pn_key)
pn_obs => parse_node_get_sub_ptr (pn_arg)
call eval_node_compile_sexpr (en0, pn_obs, var_list)
select case (res_type)
case (V_INT); var_name = "_User_obs_int"
case (V_REAL); var_name = "_User_obs_real"
end select
var => var_list_get_var_ptr (var_list, var_name, defined=.true.)
allocate (en)
if (associated (var)) then
    call eval_node_init_obs (en, var, en0)
else
    call parse_node_write (pn)
    call msg_error ("This variable is undefined at this point")
end if
if (debug) then
    call eval_node_write (en)
    print *, "done user observable"
end if
end subroutine eval_node_compile_user_observable

```

Functions with a single argument. For non-constant arguments, watch for functions which convert their argument to a different type.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_unary_function (en, pn, var_list)
type(eval_node_t), pointer :: en
type(parse_node_t), intent(in) :: pn
type(var_list_t), intent(in), target :: var_list
type(parse_node_t), pointer :: pn_fname, pn_arg
type(eval_node_t), pointer :: en1
type(string_t) :: key
integer :: t
if (debug) then
    print *, "read unary function"; call parse_node_write (pn)
end if
pn_fname => parse_node_get_sub_ptr (pn)
pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg1")
call eval_node_compile_expr &
    (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
t = en1%result_type
allocate (en)

```

```

key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT) then
  select case (char (key))
    case ("real")
      select case (t)
        case (V_INT); call eval_node_init_real (en, real_i (en1))
        case (V_REAL); deallocate (en); en => en1
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("int")
      select case (t)
        case (V_INT); deallocate (en); en => en1
        case (V_REAL); call eval_node_init_int (en, int_r (en1))
        case (V_CMPLX); call eval_node_init_int (en, int_c (en1))
      end select
    case ("nint")
      select case (t)
        case (V_INT); deallocate (en); en => en1
        case (V_REAL); call eval_node_init_int (en, nint_r (en1))
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("floor")
      select case (t)
        case (V_INT); deallocate (en); en => en1
        case (V_REAL); call eval_node_init_int (en, floor_r (en1))
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("ceiling")
      select case (t)
        case (V_INT); deallocate (en); en => en1
        case (V_REAL); call eval_node_init_int (en, ceiling_r (en1))
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("abs")
      select case (t)
        case (V_INT); call eval_node_init_int (en, abs_i (en1))
        case (V_REAL); call eval_node_init_real (en, abs_r (en1))
        case (V_CMPLX); call eval_node_init_real (en, abs_c (en1))
      end select
    case ("sgn")
      select case (t)
        case (V_INT); call eval_node_init_int (en, sgn_i (en1))
        case (V_REAL); call eval_node_init_real (en, sgn_r (en1))
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("sqrt")
      select case (t)
        case (V_REAL); call eval_node_init_real (en, sqrt_r (en1))
        case (V_CMPLX); call eval_node_init_cmplx (en, sqrt_c (en1))
        case default; call eval_type_error (pn, char (key), t)
      end select
    case ("exp")
      select case (t)
        case (V_REAL); call eval_node_init_real (en, exp_r (en1))

```

```

        case (V_CMPLX); call eval_node_init_cmplx (en, exp_c (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("log")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, log_r (en1))
        case (V_CMPLX); call eval_node_init_cmplx (en, log_c (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("log10")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, log10_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("sin")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, sin_r (en1))
        case (V_CMPLX); call eval_node_init_cmplx (en, sin_c (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("cos")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, cos_r (en1))
        case (V_CMPLX); call eval_node_init_cmplx (en, cos_c (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("tan")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, tan_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("asin")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, asin_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("acos")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, acos_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("atan")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, atan_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("sinh")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, sinh_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("cosh")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, cosh_r (en1))

```

```

        case default; call eval_type_error (pn, char (key), t)
    end select
case ("tanh")
    select case (t)
        case (V_REAL); call eval_node_init_real (en, tanh_r (en1))
        case default; call eval_type_error (pn, char (key), t)
    end select
case default
    call parse_node_mismatch ("function name", pn_fname)
end select
call eval_node_final_rec (en1)
deallocate (en1)
else
    select case (char (key))
    case ("real")
        call eval_node_init_branch (en, key, V_REAL, en1)
    case ("int", "nint", "floor", "ceiling")
        call eval_node_init_branch (en, key, V_INT, en1)
    case default
        call eval_node_init_branch (en, key, t, en1)
    end select
    select case (char (key))
    case ("real")
        select case (t)
            case (V_INT); call eval_node_set_op1_real (en, real_i)
            case (V_REAL); deallocate (en); en => en1
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("int")
        select case (t)
            case (V_INT); deallocate (en); en => en1
            case (V_REAL); call eval_node_set_op1_int (en, int_r)
            case (V_CMPLX); call eval_node_set_op1_int (en, int_c)
        end select
    case ("nint")
        select case (t)
            case (V_INT); deallocate (en); en => en1
            case (V_REAL); call eval_node_set_op1_int (en, nint_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("floor")
        select case (t)
            case (V_INT); deallocate (en); en => en1
            case (V_REAL); call eval_node_set_op1_int (en, floor_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("ceiling")
        select case (t)
            case (V_INT); deallocate (en); en => en1
            case (V_REAL); call eval_node_set_op1_int (en, ceiling_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("abs")
        select case (t)

```



```

        case (V_INT); call eval_node_set_op1_int (en, abs_i)
        case (V_REAL); call eval_node_set_op1_real (en, abs_r)
        case (V_CMPLX); call eval_node_set_op1_real (en, abs_c)
    end select
case ("sgn")
    select case (t)
        case (V_INT); call eval_node_set_op1_int (en, sgn_i)
        case (V_REAL); call eval_node_set_op1_real (en, sgn_r)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("sqrt")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, sqrt_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, sqrt_c)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("exp")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, exp_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, exp_c)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("log")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, log_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, log_c)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("log10")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, log10_r)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("sin")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, sin_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, sin_c)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("cos")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, cos_r)
        case (V_CMPLX); call eval_node_set_op1_cmplx (en, cos_c)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("tan")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, tan_r)
        case default; call eval_type_error (pn, char (key), t)
    end select
case ("asin")
    select case (t)
        case (V_REAL); call eval_node_set_op1_real (en, asin_r)
        case default; call eval_type_error (pn, char (key), t)

```

```

        end select
    case ("acos")
        select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, acos_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("atan")
        select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, atan_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("sinh")
        select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, sinh_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("cosh")
        select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, cosh_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case ("tanh")
        select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, tanh_r)
            case default; call eval_type_error (pn, char (key), t)
        end select
    case default
        call parse_node_mismatch ("function name", pn_fname)
    end select
end if
if (debug) then
    call eval_node_write (en)
    print *, "done function"
end if
end subroutine eval_node_compile_unary_function

```

Functions with two arguments.

*(Expressions: procedures)* +  $\equiv$

```

recursive subroutine eval_node_compile_binary_function (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_fname, pn_arg, pn_arg1, pn_arg2
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2
    if (debug) then
        print *, "read binary function"; call parse_node_write (pn)
    end if
    pn_fname => parse_node_get_sub_ptr (pn)
    pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg2")
    pn_arg1 => parse_node_get_sub_ptr (pn_arg, tag="expr")
    pn_arg2 => parse_node_get_next_ptr (pn_arg1, tag="expr")
    call eval_node_compile_expr (en1, pn_arg1, var_list)

```

```

call eval_node_compile_expr (en2, pn_arg2, var_list)
t1 = en1%result_type
t2 = en2%result_type
allocate (en)
key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
  select case (char (key))
  case ("max")
    select case (t1)
    case (V_INT)
      select case (t2)
      case (V_INT); call eval_node_init_int (en, max_ii (en1, en2))
      case (V_REAL); call eval_node_init_real (en, max_ir (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case (V_REAL)
      select case (t2)
      case (V_INT); call eval_node_init_real (en, max_ri (en1, en2))
      case (V_REAL); call eval_node_init_real (en, max_rr (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case default; call eval_type_error (pn, char (key), t1)
  end select
  case ("min")
    select case (t1)
    case (V_INT)
      select case (t2)
      case (V_INT); call eval_node_init_int (en, min_ii (en1, en2))
      case (V_REAL); call eval_node_init_real (en, min_ir (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case (V_REAL)
      select case (t2)
      case (V_INT); call eval_node_init_real (en, min_ri (en1, en2))
      case (V_REAL); call eval_node_init_real (en, min_rr (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case default; call eval_type_error (pn, char (key), t1)
  end select
  case ("mod")
    select case (t1)
    case (V_INT)
      select case (t2)
      case (V_INT); call eval_node_init_int (en, mod_ii (en1, en2))
      case (V_REAL); call eval_node_init_real (en, mod_ir (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case (V_REAL)
      select case (t2)
      case (V_INT); call eval_node_init_real (en, mod_ri (en1, en2))
      case (V_REAL); call eval_node_init_real (en, mod_rr (en1, en2))
      case default; call eval_type_error (pn, char (key), t2)
      end select
    case default; call eval_type_error (pn, char (key), t1)
  end select
end if

```

```

        end select
    case ("modulo")
        select case (t1)
            case (V_INT)
                select case (t2)
                    case (V_INT); call eval_node_init_int  (en, modulo_ii (en1, en2))
                    case (V_REAL); call eval_node_init_real (en, modulo_ir (en1, en2))
                    case default;  call eval_type_error (pn, char (key), t2)
                end select
            case (V_REAL)
                select case (t2)
                    case (V_INT); call eval_node_init_real (en, modulo_ri (en1, en2))
                    case (V_REAL); call eval_node_init_real (en, modulo_rr (en1, en2))
                    case default;  call eval_type_error (pn, char (key), t2)
                end select
            case default; call eval_type_error (pn, char (key), t2)
        end select
    case default
        call parse_node_mismatch ("function name", pn_fname)
    end select
    call eval_node_final_rec (en1)
    deallocate (en1)
else
    call eval_node_init_branch (en, key, t1, en1, en2)
    select case (char (key))
        case ("max")
            select case (t1)
                case (V_INT)
                    select case (t2)
                        case (V_INT); call eval_node_set_op2_int  (en, max_ii)
                        case (V_REAL); call eval_node_set_op2_real (en, max_ir)
                        case default;  call eval_type_error (pn, char (key), t2)
                    end select
                case (V_REAL)
                    select case (t2)
                        case (V_INT); call eval_node_set_op2_real (en, max_ri)
                        case (V_REAL); call eval_node_set_op2_real (en, max_rr)
                        case default;  call eval_type_error (pn, char (key), t2)
                    end select
                case default; call eval_type_error (pn, char (key), t2)
            end select
        case ("min")
            select case (t1)
                case (V_INT)
                    select case (t2)
                        case (V_INT); call eval_node_set_op2_int  (en, min_ii)
                        case (V_REAL); call eval_node_set_op2_real (en, min_ir)
                        case default;  call eval_type_error (pn, char (key), t2)
                    end select
                case (V_REAL)
                    select case (t2)
                        case (V_INT); call eval_node_set_op2_real (en, min_ri)
                        case (V_REAL); call eval_node_set_op2_real (en, min_rr)
                        case default;  call eval_type_error (pn, char (key), t2)
                    end select
            end select
        case default; call eval_type_error (pn, char (key), t2)
    end select
end select

```

```

        end select
        case default; call eval_type_error (pn, char (key), t2)
    end select
case ("mod")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_int (en, mod_ii)
        case (V_REAL); call eval_node_set_op2_real (en, mod_ir)
        case default; call eval_type_error (pn, char (key), t2)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_real (en, mod_ri)
        case (V_REAL); call eval_node_set_op2_real (en, mod_rr)
        case default; call eval_type_error (pn, char (key), t2)
        end select
    case default; call eval_type_error (pn, char (key), t2)
    end select
case ("modulo")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_int (en, modulo_ii)
        case (V_REAL); call eval_node_set_op2_real (en, modulo_ir)
        case default; call eval_type_error (pn, char (key), t2)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_real (en, modulo_ri)
        case (V_REAL); call eval_node_set_op2_real (en, modulo_rr)
        case default; call eval_type_error (pn, char (key), t2)
        end select
    case default; call eval_type_error (pn, char (key), t2)
    end select
case default
    call parse_node_mismatch ("function name", pn_fname)
end select
end if
if (debug) then
    call eval_node_write (en)
    print *, "done function"
end if
end subroutine eval_node_compile_binary_function

```

## Variable definition

A block expression contains a variable definition (first argument) and an expression where the definition can be used (second argument). The `result_type` decides which type of expression is expected for the second argument. For numeric variables, if there is a mismatch between real and integer type, insert an extra node for type conversion.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_block_expr &
    (en, pn, var_list, result_type)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in), optional :: result_type
    type(parse_node_t), pointer :: pn_var_spec, pn_var_subspec
    type(parse_node_t), pointer :: pn_var_type, pn_var_name, pn_var_expr
    type(parse_node_t), pointer :: pn_expr
    type(string_t) :: var_name
    type(eval_node_t), pointer :: en1, en2
    integer :: var_type
    logical :: new
    if (debug) then
        print *, "read block expr"; call parse_node_write (pn)
    end if
    new = .false.
    pn_var_spec => parse_node_get_sub_ptr (pn, 2)
    select case (char (parse_node_get_rule_key (pn_var_spec)))
    case ("var_num");      var_type = V_NONE
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec)
    case ("var_int");      var_type = V_INT
        new = .true.
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_real");     var_type = V_REAL
        new = .true.
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_cmplx");    var_type = V_CMPLX
        new = .true.
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_logical_new"); var_type = V_LOG
        new = .true.
        pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
        pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
    case ("var_logical_spec"); var_type = V_LOG
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_plist_new"); var_type = V_SEV
        new = .true.
        pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
        pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
    case ("var_plist_spec"); var_type = V_SEV
        new = .true.
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_alias");    var_type = V_PDG
        new = .true.
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case ("var_string_new"); var_type = V_STR
        new = .true.
        pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
        pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
    case ("var_string_spec"); var_type = V_STR
        pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case default

```

```

        call parse_node_mismatch &
            ("logical|int|real|plist|alias", pn_var_type)
    end select
    pn_var_expr => parse_node_get_next_ptr (pn_var_name, 2)
    pn_expr => parse_node_get_next_ptr (pn_var_spec, 2)
    var_name = parse_node_get_string (pn_var_name)
    select case (var_type)
    case (V_LOG); var_name = "?" // var_name
    case (V_SEV); var_name = "@" // var_name
    case (V_STR); var_name = "$" // var_name      ! $ sign
    end select
    call var_list_check_user_var (var_list, var_name, var_type, new)
    call eval_node_compile_genexpr (en1, pn_var_expr, var_list, var_type)
    call insert_conversion_node (en1, var_type)
    allocate (en)
    call eval_node_init_block (en, var_name, var_type, en1, var_list)
    call eval_node_compile_genexpr (en2, pn_expr, en%var_list, result_type)
    call eval_node_set_expr (en, en2)
    if (debug) then
        call eval_node_write (en)
        print *, "done block expr"
    end if
end subroutine eval_node_compile_block_expr

```

Insert a conversion node for integer/real/complex transformation if necessary.  
 What shall we do for the complex to integer/real conversion?

*(Expressions: procedures)*+≡

```

subroutine insert_conversion_node (en, result_type)
    type(eval_node_t), pointer :: en
    integer, intent(in) :: result_type
    type(eval_node_t), pointer :: en_conv
    select case (en%result_type)
    case (V_INT)
        select case (result_type)
        case (V_REAL)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
            call eval_node_set_op1_real (en_conv, real_i)
            en => en_conv
        case (V_CMPLX)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
            call eval_node_set_op1_cmplx (en_conv, cmplx_i)
            en => en_conv
        end select
    case (V_REAL)
        select case (result_type)
        case (V_INT)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
            call eval_node_set_op1_int (en_conv, int_r)
            en => en_conv
        case (V_CMPLX)

```

```

        allocate (en_conv)
        call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
        call eval_node_set_op1_cmplx (en_conv, cmplx_r)
        en => en_conv
    end select
case (V_CMPLX)
    select case (result_type)
    case (V_INT)
        allocate (en_conv)
        call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
        call eval_node_set_op1_int (en_conv, int_c)
        en => en_conv
    case (V_REAL)
        allocate (en_conv)
        call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
        call eval_node_set_op1_real (en_conv, real_c)
        en => en_conv
    end select
case default
end select
end subroutine insert_conversion_node

```

## Conditionals

A conditional has the structure `if lexpr then expr else expr`. So we first evaluate the logical expression, then depending on the result the first or second expression. Note that the second expression is mandatory.

The `result_type`, if present, defines the requested type of the `then` and `else` clauses. Default is numeric (int/real). If there is a mismatch between real and integer result types, insert conversion nodes.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_conditional &
    (en, pn, var_list, result_type)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in), optional :: result_type
    type(parse_node_t), pointer :: pn_condition, pn_expr
    type(parse_node_t), pointer :: pn_maybe_elseif, pn_elseif_branch
    type(parse_node_t), pointer :: pn_maybe_else, pn_else_branch, pn_else_expr
    type(eval_node_t), pointer :: en0, en1, en2
    integer :: restype
    if (debug) then
        print *, "read conditional"; call parse_node_write (pn)
    end if
    pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
    pn_expr => parse_node_get_next_ptr (pn_condition, 2)
    call eval_node_compile_lexpr (en0, pn_condition, var_list)
    call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
    if (present (result_type)) then
        restype = major_result_type (result_type, en1%result_type)
    else

```



```

        restype = en1%result_type
    end if
    pn_maybe_elseif => parse_node_get_next_ptr (pn_expr)
    select case (char (parse_node_get_rule_key (pn_maybe_elseif)))
    case ("maybe_elseif_expr", &
        "maybe_elseif_lexpr", &
        "maybe_elseif_pexpr", &
        "maybe_elseif_cexpr", &
        "maybe_elseif_sexpr")
        pn_elseif_branch => parse_node_get_sub_ptr (pn_maybe_elseif)
        pn_maybe_else => parse_node_get_next_ptr (pn_maybe_elseif)
        select case (char (parse_node_get_rule_key (pn_maybe_else)))
        case ("maybe_else_expr", &
            "maybe_else_lexpr", &
            "maybe_else_pexpr", &
            "maybe_else_cexpr", &
            "maybe_else_sexpr")
            pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
            pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
        case default
            pn_else_expr => null ()
        end select
        call eval_node_compile_elseif &
            (en2, pn_elseif_branch, pn_else_expr, var_list, restype)
    case ("maybe_else_expr", &
        "maybe_else_lexpr", &
        "maybe_else_pexpr", &
        "maybe_else_cexpr", &
        "maybe_else_sexpr")
        pn_maybe_else => pn_maybe_elseif
        pn_maybe_elseif => null ()
        pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
        pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
        call eval_node_compile_genexpr &
            (en2, pn_else_expr, var_list, restype)
    case ("endif")
        call eval_node_compile_default_else (en2, restype)
    case default
        call msg_bug ("Broken conditional: unexpected " &
            // char (parse_node_get_rule_key (pn_maybe_elseif)))
    end select
    call eval_node_create_conditional (en, en0, en1, en2, restype)
    call conditional_insert_conversion_nodes (en, restype)
    if (debug) then
        call eval_node_write (en)
        print *, "done conditional"
    end if
end subroutine eval_node_compile_conditional

```

This recursively generates 'elseif' conditionals as a chain of sub-nodes of the main conditional.

$\langle \text{Expressions: procedures} \rangle + \equiv$

recursive subroutine eval\_node\_compile\_elseif &

```

      (en, pn, pn_else_expr, var_list, result_type)
type(eval_node_t), pointer :: en
type(parse_node_t), intent(in), target :: pn
type(parse_node_t), pointer :: pn_else_expr
type(var_list_t), intent(in), target :: var_list
integer, intent(inout) :: result_type
type(parse_node_t), pointer :: pn_next, pn_condition, pn_expr
type(eval_node_t), pointer :: en0, en1, en2
pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
pn_expr => parse_node_get_next_ptr (pn_condition, 2)
call eval_node_compile_lexpr (en0, pn_condition, var_list)
call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
result_type = major_result_type (result_type, en1%result_type)
pn_next => parse_node_get_next_ptr (pn)
if (associated (pn_next)) then
  call eval_node_compile_elseif &
    (en2, pn_next, pn_else_expr, var_list, result_type)
  result_type = major_result_type (result_type, en2%result_type)
else if (associated (pn_else_expr)) then
  call eval_node_compile_genexpr &
    (en2, pn_else_expr, var_list, result_type)
  result_type = major_result_type (result_type, en2%result_type)
else
  call eval_node_compile_default_else (en2, result_type)
end if
call eval_node_create_conditional (en, en0, en1, en2, result_type)
end subroutine eval_node_compile_elseif

```

This makes a default 'else' branch in case it was omitted. The default value just depends on the expected type.

*(Expressions: procedures)*+≡

```

subroutine eval_node_compile_default_else (en, result_type)
type(eval_node_t), pointer :: en
integer, intent(in) :: result_type
type(subvt_t) :: pval_empty
type(pdg_array_t) :: aval_undefined
allocate (en)
select case (result_type)
case (V_LOG); call eval_node_init_log (en, .false.)
case (V_INT); call eval_node_init_int (en, 0)
case (V_REAL); call eval_node_init_real (en, 0._default)
case (V_CMPLX)
  call eval_node_init_cmplx (en, (0._default, 0._default))
case (V_SEV)
  call subvt_init (pval_empty)
  call eval_node_init_subvt (en, pval_empty)
case (V_PDG)
  call eval_node_init_pdg_array (en, aval_undefined)
case (V_STR)
  call eval_node_init_string (en, var_str (""))
case default
  call msg_bug ("Undefined type for 'else' branch in conditional")
end select

```

```
end subroutine eval_node_compile_default_else
```

If the logical expression is constant, we can simplify the conditional node by replacing it with the selected branch. Otherwise, we initialize a true branching.

*(Expressions: procedures)+≡*

```
subroutine eval_node_create_conditional (en, en0, en1, en2, result_type)
  type(eval_node_t), pointer :: en, en0, en1, en2
  integer, intent(in) :: result_type
  if (en0%type == EN_CONSTANT) then
    if (en0%lval) then
      en => en1
      call eval_node_final_rec (en2)
      deallocate (en2)
    else
      en => en2
      call eval_node_final_rec (en1)
      deallocate (en1)
    end if
  else
    allocate (en)
    call eval_node_init_conditional (en, result_type, en0, en1, en2)
  end if
end subroutine eval_node_create_conditional
```

Return the numerical result type which should be used for the combination of the two result types.

*(Expressions: procedures)+≡*

```
function major_result_type (t1, t2) result (t)
  integer :: t
  integer, intent(in) :: t1, t2
  select case (t1)
  case (V_INT)
    select case (t2)
    case (V_INT, V_REAL, V_CMPLX)
      t = t2
    case default
      call type_mismatch ()
    end select
  case (V_REAL)
    select case (t2)
    case (V_INT)
      t = t1
    case (V_REAL, V_CMPLX)
      t = t2
    case default
      call type_mismatch ()
    end select
  case (V_CMPLX)
    select case (t2)
    case (V_INT, V_REAL, V_CMPLX)
      t = t1
    case default
      call type_mismatch ()
    end select
  end select
```

```

        end select
    case default
        if (t1 == t2) then
            t = t1
        else
            call type_mismatch ()
        end if
    end select
contains
    subroutine type_mismatch ()
        call msg_bug ("Type mismatch in branches of a conditional expression")
    end subroutine type_mismatch
end function major_result_type

```

Recursively insert conversion nodes where necessary.

*(Expressions: procedures)*+≡

```

recursive subroutine conditional_insert_conversion_nodes (en, result_type)
    type(eval_node_t), intent(inout), target :: en
    integer, intent(in) :: result_type
    select case (result_type)
    case (V_INT, V_REAL, V_CMPLX)
        call insert_conversion_node (en%arg1, result_type)
        if (en%arg2%type == EN_CONDITIONAL) then
            call conditional_insert_conversion_nodes (en%arg2, result_type)
        else
            call insert_conversion_node (en%arg2, result_type)
        end if
    end select
end subroutine conditional_insert_conversion_nodes

```

## Logical expressions

A logical expression consists of one or more singlet logical expressions concatenated by ;. This is for allowing side-effects, only the last value is used.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_lexpr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_term, pn_sequel, pn_arg
    type(eval_node_t), pointer :: en1, en2
    if (debug) then
        print *, "read lexpr"; call parse_node_write (pn)
    end if
    pn_term => parse_node_get_sub_ptr (pn, tag="lsinglet")
    call eval_node_compile_lsinglet (en, pn_term, var_list)
    pn_sequel => parse_node_get_next_ptr (pn_term, tag="lsequel")
    do while (associated (pn_sequel))
        pn_arg => parse_node_get_sub_ptr (pn_sequel, 2, tag="lsinglet")
        en1 => en
        call eval_node_compile_lsinglet (en2, pn_arg, var_list)
        allocate (en)
    end do
end subroutine eval_node_compile_lexpr

```

```

        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call eval_node_init_log (en, ignore_first_ll (en1, en2))
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
        else
            call eval_node_init_branch &
                (en, var_str ("lsequel"), V_LOG, en1, en2)
            call eval_node_set_op2_log (en, ignore_first_ll)
        end if
        pn_sequel => parse_node_get_next_ptr (pn_sequel)
    end do
    if (debug) then
        call eval_node_write (en)
        print *, "done lexpr"
    end if
end subroutine eval_node_compile_lexpr

```

A logical singlet expression consists of one or more logical terms concatenated by or.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_lsinglet (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_term, pn_alternative, pn_arg
    type(eval_node_t), pointer :: en1, en2
    if (debug) then
        print *, "read lsinglet"; call parse_node_write (pn)
    end if
    pn_term => parse_node_get_sub_ptr (pn, tag="lterm")
    call eval_node_compile_lterm (en, pn_term, var_list)
    pn_alternative => parse_node_get_next_ptr (pn_term, tag="alternative")
    do while (associated (pn_alternative))
        pn_arg => parse_node_get_sub_ptr (pn_alternative, 2, tag="lterm")
        en1 => en
        call eval_node_compile_lterm (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call eval_node_init_log (en, or_ll (en1, en2))
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
        else
            call eval_node_init_branch &
                (en, var_str ("alternative"), V_LOG, en1, en2)
            call eval_node_set_op2_log (en, or_ll)
        end if
        pn_alternative => parse_node_get_next_ptr (pn_alternative)
    end do
    if (debug) then
        call eval_node_write (en)
        print *, "done lsinglet"
    end if
end subroutine eval_node_compile_lsinglet

```

```

end if
end subroutine eval_node_compile_lsinglet

```

A logical term consists of one or more logical values concatenated by **and**.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_lterm (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_term, pn_coincidence, pn_arg
  type(eval_node_t), pointer :: en1, en2
  if (debug) then
    print *, "read lterm"; call parse_node_write (pn)
  end if
  pn_term => parse_node_get_sub_ptr (pn)
  call eval_node_compile_lvalue (en, pn_term, var_list)
  pn_coincidence => parse_node_get_next_ptr (pn_term, tag="coincidence")
  do while (associated (pn_coincidence))
    pn_arg => parse_node_get_sub_ptr (pn_coincidence, 2)
    en1 => en
    call eval_node_compile_lvalue (en2, pn_arg, var_list)
    allocate (en)
    if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
      call eval_node_init_log (en, and_ll (en1, en2))
      call eval_node_final_rec (en1)
      call eval_node_final_rec (en2)
      deallocate (en1, en2)
    else
      call eval_node_init_branch &
        (en, var_str ("coincidence"), V_LOG, en1, en2)
      call eval_node_set_op2_log (en, and_ll)
    end if
    pn_coincidence => parse_node_get_next_ptr (pn_coincidence)
  end do
  if (debug) then
    call eval_node_write (en)
    print *, "done lterm"
  end if
end subroutine eval_node_compile_lterm

```

Logical variables are disabled, because they are confused with the l.h.s. of compared expressions.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_lvalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  if (debug) then
    print *, "read lvalue"; call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("true")
    allocate (en)

```

```

        call eval_node_init_log (en, .true.)
case ("false")
    allocate (en)
    call eval_node_init_log (en, .false.)
case ("negation")
    call eval_node_compile_negation (en, pn, var_list)
case ("lvariable")
    call eval_node_compile_variable (en, pn, var_list, V_LOG)
case ("lexpr")
    call eval_node_compile_lexpr (en, pn, var_list)
case ("block_lexpr")
    call eval_node_compile_block_expr (en, pn, var_list, V_LOG)
case ("conditional_lexpr")
    call eval_node_compile_conditional (en, pn, var_list, V_LOG)
case ("compared_expr")
    call eval_node_compile_compared_expr (en, pn, var_list, V_REAL)
case ("compared_sexpr")
    call eval_node_compile_compared_expr (en, pn, var_list, V_STR)
case ("all_fun", "any_fun", "no_fun", "user_cut_fun")
    call eval_node_compile_log_function (en, pn, var_list)
case ("record_cmd")
    call eval_node_compile_record_cmd (en, pn, var_list)
case default
    call parse_node_mismatch &
        ("true|false|negation|lvariable|" // &
        "lexpr|block_lexpr|conditional_lexpr|" // &
        "compared_expr|compared_sexpr|logical_pexpr", pn)
end select
if (debug) then
    call eval_node_write (en)
    print *, "done lvalue"
end if
end subroutine eval_node_compile_lvalue

```

A negation consists of the keyword not and a logical value.

*(Expressions: procedures)* +=

```

recursive subroutine eval_node_compile_negation (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_arg
    type(eval_node_t), pointer :: en1
    if (debug) then
        print *, "read negation"; call parse_node_write (pn)
    end if
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    call eval_node_compile_lvalue (en1, pn_arg, var_list)
    allocate (en)
    if (en1%type == EN_CONSTANT) then
        call eval_node_init_log (en, not_1 (en1))
        call eval_node_final_rec (en1)
        deallocate (en1)
    else
        call eval_node_init_branch (en, var_str ("not"), V_LOG, en1)
    end if
end subroutine eval_node_compile_negation

```

```

        call eval_node_set_op1_log (en, not_1)
    end if
    if (debug) then
        call eval_node_write (en)
        print *, "done negation"
    end if
end subroutine eval_node_compile_negation

```

## Comparisons

Up to the loop, this is easy. There is always at least one comparison. This is evaluated, and the result is the logical node `en`. If it is constant, we keep its second sub-node as `en2`. (Thus, at the very end `en2` has to be deleted if `en` is (still) constant.)

If there is another comparison, we first check if the first comparison was constant. In that case, there are two possibilities: (i) it was true. Then, its right-hand side is compared with the new right-hand side, and the result replaces the previous one which is deleted. (ii) it was false. In this case, the result of the whole comparison is false, and we can exit the loop without evaluating anything else.

Now assume that the first comparison results in a valid branch, its second sub-node kept as `en2`. We first need a copy of this, which becomes the new left-hand side. If `en2` is constant, we make an identical constant node `en1`. Otherwise, we make `en1` an appropriate pointer node. Next, the first branch is saved as `en0` and we evaluate the comparison between `en1` and the a right-hand side. If this turns out to be constant, there are again two possibilities: (i) true, then we revert to the previous result. (ii) false, then the wh

*(Expressions: procedures)+≡*

```

recursive subroutine eval_node_compile_compared_expr (en, pn, var_list, type)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in) :: type
    type(parse_node_t), pointer :: pn_comparison, pn_expr1
    type(eval_node_t), pointer :: en0, en1, en2
    if (debug) then
        print *, "read comparison"; call parse_node_write (pn)
    end if
    select case (type)
    case (V_INT, V_REAL)
        pn_expr1 => parse_node_get_sub_ptr (pn, tag="expr")
        call eval_node_compile_expr (en1, pn_expr1, var_list)
        pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="comparison")
    case (V_STR)
        pn_expr1 => parse_node_get_sub_ptr (pn, tag="sexpr")
        call eval_node_compile_sexpr (en1, pn_expr1, var_list)
        pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="str_comparison")
    end select
    call eval_node_compile_comparison &
        (en, en1, en2, pn_comparison, var_list, type)
    pn_comparison => parse_node_get_next_ptr (pn_comparison)

```



```

SCAN_FURTHER: do while (associated (pn_comparation))
  if (en%type == EN_CONSTANT) then
    if (en%lval) then
      en1 => en2
      call eval_node_final_rec (en); deallocate (en)
      call eval_node_compile_comparation &
        (en, en1, en2, pn_comparation, var_list, type)
    else
      exit SCAN_FURTHER
    end if
  else
    allocate (en1)
    if (en2%type == EN_CONSTANT) then
      select case (en2%result_type)
        case (V_INT); call eval_node_init_int      (en1, en2%ival)
        case (V_REAL); call eval_node_init_real   (en1, en2%rval)
        case (V_STR); call eval_node_init_string (en1, en2%sval)
      end select
    else
      select case (en2%result_type)
        case (V_INT); call eval_node_init_int_ptr &
          (en1, var_str ("(previous)"), en2%ival, en2%value_is_known)
        case (V_REAL); call eval_node_init_real_ptr &
          (en1, var_str ("(previous)"), en2%rval, en2%value_is_known)
        case (V_STR); call eval_node_init_string_ptr &
          (en1, var_str ("(previous)"), en2%sval, en2%value_is_known)
      end select
    end if
    en0 => en
    call eval_node_compile_comparation &
      (en, en1, en2, pn_comparation, var_list, type)
    if (en%type == EN_CONSTANT) then
      if (en%lval) then
        call eval_node_final_rec (en); deallocate (en)
        en => en0
      else
        call eval_node_final_rec (en0); deallocate (en0)
        exit SCAN_FURTHER
      end if
    else
      en1 => en
      allocate (en)
      call eval_node_init_branch (en, var_str ("and"), V_LOG, en0, en1)
      call eval_node_set_op2_log (en, and_ll)
    end if
  end if
  pn_comparation => parse_node_get_next_ptr (pn_comparation)
end do SCAN_FURTHER
if (en%type == EN_CONSTANT .and. associated (en2)) then
  call eval_node_final_rec (en2); deallocate (en2)
end if
if (debug) then
  call eval_node_write (en)
  print *, "done compared_expr"

```

```

end if
end subroutine eval_node_compile_compared_expr

```

This takes two extra arguments: **en1**, the left-hand-side of the comparison, is already allocated and evaluated. **en2** (the right-hand side) and **en** (the result) are allocated by the routine. **pn** is the parse node which contains the operator and the right-hand side as subnodes.

If the result of the comparison is constant, **en1** is deleted but **en2** is kept, because it may be used in a subsequent comparison. **en** then becomes a constant. If the result is variable, **en** becomes a branch node which refers to **en1** and **en2**.

(*Expressions: procedures*) +=

```

recursive subroutine eval_node_compile_comparison &
  (en, en1, en2, pn, var_list, type)
  type(eval_node_t), pointer :: en, en1, en2
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in) :: type
  type(parse_node_t), pointer :: pn_op, pn_arg
  type(string_t) :: key
  integer :: t1, t2
  type(var_entry_t), pointer :: var
  pn_op => parse_node_get_sub_ptr (pn)
  key = parse_node_get_key (pn_op)
  select case (type)
  case (V_INT, V_REAL)
    pn_arg => parse_node_get_next_ptr (pn_op, tag="expr")
    call eval_node_compile_expr (en2, pn_arg, var_list)
  case (V_STR)
    pn_arg => parse_node_get_next_ptr (pn_op, tag="sexpr")
    call eval_node_compile_sexpr (en2, pn_arg, var_list)
  end select
  t1 = en1%result_type
  t2 = en2%result_type
  allocate (en)
  if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
    select case (char (key))
    case ("<")
      select case (t1)
      case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_init_log (en, comp_lt_ii (en1, en2))
        case (V_REAL); call eval_node_init_log (en, comp_lt_ir (en1, en2))
        end select
      case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_init_log (en, comp_lt_ri (en1, en2))
        case (V_REAL); call eval_node_init_log (en, comp_lt_rr (en1, en2))
        end select
      end select
    case (">")
      select case (t1)
      case (V_INT)
        select case (t2)

```

```

        case (V_INT); call eval_node_init_log (en, comp_gt_ii (en1, en2))
        case (V_REAL); call eval_node_init_log (en, comp_gt_ir (en1, en2))
    end select
case (V_REAL)
    select case (t2)
        case (V_INT); call eval_node_init_log (en, comp_gt_ri (en1, en2))
        case (V_REAL); call eval_node_init_log (en, comp_gt_rr (en1, en2))
    end select
end select
case ("<=")
    select case (t1)
        case (V_INT)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_le_ii (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_le_ir (en1, en2))
            end select
        case (V_REAL)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_le_ri (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_le_rr (en1, en2))
            end select
        end select
    end select
case (">=")
    select case (t1)
        case (V_INT)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_ge_ii (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_ge_ir (en1, en2))
            end select
        case (V_REAL)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_ge_ri (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_ge_rr (en1, en2))
            end select
        end select
    end select
case ("==")
    select case (t1)
        case (V_INT)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_eq_ii (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_eq_ir (en1, en2))
            end select
        case (V_REAL)
            select case (t2)
                case (V_INT); call eval_node_init_log (en, comp_eq_ri (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_eq_rr (en1, en2))
            end select
        case (V_STR)
            select case (t2)
                case (V_STR); call eval_node_init_log (en, comp_eq_ss (en1, en2))
            end select
        end select
    end select
case ("<>")
    select case (t1)

```

```

case (V_INT)
  select case (t2)
    case (V_INT); call eval_node_init_log (en, comp_ne_ii (en1, en2))
    case (V_REAL); call eval_node_init_log (en, comp_ne_ir (en1, en2))
  end select
case (V_REAL)
  select case (t2)
    case (V_INT); call eval_node_init_log (en, comp_ne_ri (en1, en2))
    case (V_REAL); call eval_node_init_log (en, comp_ne_rr (en1, en2))
  end select
case (V_STR)
  select case (t2)
    case (V_STR); call eval_node_init_log (en, comp_ne_ss (en1, en2))
  end select
end select
case ("==" )
var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
en1%tolerance => var_entry_get_rval_ptr (var)
select case (t1)
case (V_INT)
  select case (t2)
    case (V_INT); call eval_node_init_log (en, comp_sim_ii (en1, en2))
    case (V_REAL); call eval_node_init_log (en, comp_sim_ir (en1, en2))
  end select
case (V_REAL)
  select case (t2)
    case (V_INT); call eval_node_init_log (en, comp_sim_ri (en1, en2))
    case (V_REAL); call eval_node_init_log (en, comp_sim_rr (en1, en2))
  end select
case (V_STR)
  select case (t2)
    case (V_STR); call eval_node_init_log (en, comp_eq_ss (en1, en2))
  end select
end select
case ("<>" )
var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
en1%tolerance => var_entry_get_rval_ptr (var)
select case (t1)
case (V_INT)
  select case (t2)
    case (V_INT)
      call eval_node_init_log (en, comp_nsim_ii (en1, en2))
    case (V_REAL)
      call eval_node_init_log (en, comp_nsim_ir (en1, en2))
  end select
case (V_REAL)
  select case (t2)
    case (V_INT)
      call eval_node_init_log (en, comp_nsim_ri (en1, en2))
    case (V_REAL)
      call eval_node_init_log (en, comp_nsim_rr(en1, en2))
  end select
case (V_STR)
  select case (t2)

```

```

        case (V_STR); call eval_node_init_log (en, comp_ne_ss (en1, en2))
    end select
end select
end select
call eval_node_final_rec (en1)
deallocate (en1)
else
call eval_node_init_branch (en, key, V_LOG, en1, en2)
select case (char (key))
case ("<")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_lt_ii)
        case (V_REAL); call eval_node_set_op2_log (en, comp_lt_ir)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_lt_ri)
        case (V_REAL); call eval_node_set_op2_log (en, comp_lt_rr)
        end select
    end select
case (">")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_gt_ii)
        case (V_REAL); call eval_node_set_op2_log (en, comp_gt_ir)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_gt_ri)
        case (V_REAL); call eval_node_set_op2_log (en, comp_gt_rr)
        end select
    end select
case ("<=")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_le_ii)
        case (V_REAL); call eval_node_set_op2_log (en, comp_le_ir)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_le_ri)
        case (V_REAL); call eval_node_set_op2_log (en, comp_le_rr)
        end select
    end select
case (">=")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_ge_ii)
        case (V_REAL); call eval_node_set_op2_log (en, comp_ge_ir)

```

```

        end select
    case (V_REAL)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_ge_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_ge_rr)
        end select
    end select
case ("==")
    select case (t1)
    case (V_INT)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_eq_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_eq_ir)
        end select
    case (V_REAL)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_eq_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_eq_rr)
        end select
    case (V_STR)
        select case (t2)
            case (V_STR); call eval_node_set_op2_log (en, comp_eq_ss)
        end select
    end select
case ("<>")
    select case (t1)
    case (V_INT)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_ne_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_ne_ir)
        end select
    case (V_REAL)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_ne_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_ne_rr)
        end select
    case (V_STR)
        select case (t2)
            case (V_STR); call eval_node_set_op2_log (en, comp_ne_ss)
        end select
    end select
case ("==~")
    select case (t1)
    case (V_INT)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_sim_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_sim_ir)
        end select
    case (V_REAL)
        select case (t2)
            case (V_INT); call eval_node_set_op2_log (en, comp_sim_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_sim_rr)
        end select
    case (V_STR)

```

```

        select case (t2)
        case (V_STR); call eval_node_set_op2_log (en, comp_eq_ss)
        end select
    end select
    var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
    en1%tolerance => var_entry_get_rval_ptr (var)
case ("<>~")
    select case (t1)
    case (V_INT)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_nsim_ii)
        case (V_REAL); call eval_node_set_op2_log (en, comp_nsim_ir)
        end select
    case (V_REAL)
        select case (t2)
        case (V_INT); call eval_node_set_op2_log (en, comp_nsim_ri)
        case (V_REAL); call eval_node_set_op2_log (en, comp_nsim_rr)
        end select
    case (V_STR)
        select case (t2)
        case (V_STR); call eval_node_set_op2_log (en, comp_ne_ss)
        end select
    end select
    var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
    en1%tolerance => var_entry_get_rval_ptr (var)
    end select
end if
end subroutine eval_node_compile_comparison

```

## Recording analysis data

The `record` command is actually a logical expression which always evaluates true.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_record_cmd (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_key, pn_tag, pn_arg
    type(parse_node_t), pointer :: pn_arg1, pn_arg2, pn_arg3, pn_arg4
    type(eval_node_t), pointer :: en0, en1, en2, en3, en4
    type(var_entry_t), pointer :: var
    real(default), pointer :: event_weight
    if (debug) then
        print *, "read record_cmd"; call parse_node_write (pn)
    end if
    pn_key => parse_node_get_sub_ptr (pn)
    pn_tag => parse_node_get_next_ptr (pn_key)
    pn_arg => parse_node_get_next_ptr (pn_tag)
    select case (char (parse_node_get_key (pn_key)))
    case ("record")
        var => var_list_get_var_ptr (var_list, var_str ("event_weight"))

```

```

        if (associated (var)) then
            event_weight => var_entry_get_rval_ptr (var)
        else
            event_weight => null ()
        end if
    case ("record_unweighted")
        event_weight => null ()
    end select
select case (char (parse_node_get_rule_key (pn_tag)))
case ("analysis_id")
    allocate (en0)
    call eval_node_init_string (en0, parse_node_get_string (pn_tag))
case default
    call eval_node_compile_sexpr (en0, pn_tag, var_list)
end select
allocate (en)
if (associated (pn_arg)) then
    pn_arg1 => parse_node_get_sub_ptr (pn_arg)
    call eval_node_compile_expr (en1, pn_arg1, var_list)
    if (en1%result_type == V_INT) &
        call insert_conversion_node (en1, V_REAL)
    pn_arg2 => parse_node_get_next_ptr (pn_arg1)
    if (associated (pn_arg2)) then
        call eval_node_compile_expr (en2, pn_arg2, var_list)
        if (en2%result_type == V_INT) &
            call insert_conversion_node (en2, V_REAL)
        pn_arg3 => parse_node_get_next_ptr (pn_arg2)
        if (associated (pn_arg3)) then
            call eval_node_compile_expr (en3, pn_arg3, var_list)
            if (en3%result_type == V_INT) &
                call insert_conversion_node (en3, V_REAL)
            pn_arg4 => parse_node_get_next_ptr (pn_arg3)
            if (associated (pn_arg4)) then
                call eval_node_compile_expr (en4, pn_arg4, var_list)
                if (en4%result_type == V_INT) &
                    call insert_conversion_node (en4, V_REAL)
                call eval_node_init_record_cmd &
                    (en, event_weight, en0, en1, en2, en3, en4)
            else
                call eval_node_init_record_cmd &
                    (en, event_weight, en0, en1, en2, en3)
            end if
        else
            call eval_node_init_record_cmd (en, event_weight, en0, en1, en2)
        end if
    else
        call eval_node_init_record_cmd (en, event_weight, en0, en1)
    end if
end if
else
    call eval_node_init_record_cmd (en, event_weight, en0)
end if
if (debug) then
    call eval_node_write (en)
    print *, "done record_cmd"

```



```

end if
end subroutine eval_node_compile_record_cmd

```

## Particle-list expressions

A particle expression is a subevent or a concatenation of particle-list terms (using join).

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_pexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_pterm, pn_concatenation, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(subvt_t) :: subvt
  if (debug) then
    print *, "read pexpr"; call parse_node_write (pn)
  end if
  pn_pterm => parse_node_get_sub_ptr (pn)
  call eval_node_compile_pterm (en, pn_pterm, var_list)
  pn_concatenation => &
    parse_node_get_next_ptr (pn_pterm, tag="pconcatenation")
  do while (associated (pn_concatenation))
    pn_op => parse_node_get_sub_ptr (pn_concatenation)
    pn_arg => parse_node_get_next_ptr (pn_op)
    en1 => en
    call eval_node_compile_pterm (en2, pn_arg, var_list)
    allocate (en)
    if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
      call subvt_join (subvt, en1%pval, en2%pval)
      call eval_node_init_subvt (en, subvt)
      call eval_node_final_rec (en1)
      call eval_node_final_rec (en2)
      deallocate (en1, en2)
    else
      call eval_node_init_branch &
        (en, var_str ("join"), V_SEV, en1, en2)
      call eval_node_set_op2_sev (en, join_pp)
    end if
    pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
  end do
  if (debug) then
    call eval_node_write (en)
    print *, "done pexpr"
  end if
end subroutine eval_node_compile_pexpr

```

A particle term is a subevent or a combination of particle-list values (using combine).

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_pterm (en, pn, var_list)
  type(eval_node_t), pointer :: en

```

```

type(parse_node_t), intent(in) :: pn
type(var_list_t), intent(in), target :: var_list
type(parse_node_t), pointer :: pn_pvalue, pn_combination, pn_op, pn_arg
type(eval_node_t), pointer :: en1, en2
type(subevt_t) :: subevt
if (debug) then
  print *, "read pterm"; call parse_node_write (pn)
end if
pn_pvalue => parse_node_get_sub_ptr (pn)
call eval_node_compile_pvalue (en, pn_pvalue, var_list)
pn_combination => &
  parse_node_get_next_ptr (pn_pvalue, tag="pcombination")
do while (associated (pn_combination))
  pn_op => parse_node_get_sub_ptr (pn_combination)
  pn_arg => parse_node_get_next_ptr (pn_op)
  en1 => en
  call eval_node_compile_pvalue (en2, pn_arg, var_list)
  allocate (en)
  if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
    call subevt_combine (subevt, en1%pval, en2%pval)
    call eval_node_init_subevt (en, subevt)
    call eval_node_final_rec (en1)
    call eval_node_final_rec (en2)
    deallocate (en1, en2)
  else
    call eval_node_init_branch &
      (en, var_str ("combine"), V_SEV, en1, en2)
    call eval_node_set_op2_sev (en, combine_pp)
  end if
  pn_combination => parse_node_get_next_ptr (pn_combination)
end do
if (debug) then
  call eval_node_write (en)
  print *, "done pterm"
end if
end subroutine eval_node_compile_pterm

```

A particle-list value is a PDG-code array, a particle identifier, a variable, a (grouped) pexpr, a block pexpr, a conditional, or a particle-list function.

The `cexpr` node is responsible for transforming a constant PDG-code array into a subevent. It takes the code array as its first argument, the event subevent as its second argument, and the requested particle type (incoming/outgoing) as its zero-th argument. The result is the list of particles in the event that match the code array.

*(Expressions: procedures)+≡*

```

recursive subroutine eval_node_compile_pvalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_prefix_cexpr
  type(eval_node_t), pointer :: en1, en2, en0
  type(string_t) :: key
  type(var_entry_t), pointer :: var

```

```

logical, save, target :: known = .true.
if (debug) then
  print *, "read pvalue"; call parse_node_write (pn)
end if
select case (char (parse_node_get_rule_key (pn)))
case ("pexpr_src")
  call eval_node_compile_prefix_cexpr (en1, pn, var_list)
  allocate (en2)
  var => var_list_get_var_ptr (var_list, var_str ("%evt"))
  if (associated (var)) then
    call eval_node_init_subevt_ptr &
      (en2, var_str ("%evt"), var_entry_get_pval_ptr (var), known)
    allocate (en)
    call eval_node_init_branch &
      (en, var_str ("prt_selection"), V_SEV, en1, en2)
    call eval_node_set_op2_sev (en, select_pdg_ca)
    allocate (en0)
    pn_prefix_cexpr => parse_node_get_sub_ptr (pn)
    key = parse_node_get_rule_key (pn_prefix_cexpr)
    select case (char (key))
    case ("incoming_prt")
      call eval_node_init_int (en0, PRT_INCOMING)
      en%arg0 => en0
    case ("outgoing_prt")
      call eval_node_init_int (en0, PRT_OUTGOING)
      en%arg0 => en0
    end select
  else
    call parse_node_write (pn)
    call msg_bug (" Missing event data while compiling pvalue")
  end if
case ("pvariable")
  call eval_node_compile_variable (en, pn, var_list, V_SEV)
case ("pexpr")
  call eval_node_compile_pexpr (en, pn, var_list)
case ("block_pexpr")
  call eval_node_compile_block_expr (en, pn, var_list, V_SEV)
case ("conditional_pexpr")
  call eval_node_compile_conditional (en, pn, var_list, V_SEV)
case ("join_fun", "combine_fun", "collect_fun", "select_fun", &
  "extract_fun", "sort_fun")
  call eval_node_compile_prt_function (en, pn, var_list)
case default
  call parse_node_mismatch &
    ("prefix_cexpr|pvariable|" // &
    "grouped_pexpr|block_pexpr|conditional_pexpr|" // &
    "prt_function", pn)
end select
if (debug) then
  call eval_node_write (en)
  print *, "done pvalue"
end if
end subroutine eval_node_compile_pvalue

```

## Particle functions

This combines the treatment of 'join', 'combine', 'collect', 'select', and 'extract' which all have the same syntax. The one or two argument nodes are allocated. If there is a condition, the condition node is also allocated as a logical expression, for which the variable list is augmented by the appropriate (unary/binary) observables.

*(Expressions: procedures)* +≡

```
recursive subroutine eval_node_compile_prt_function (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
  type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en0, en1, en2
  type(string_t) :: key
  if (debug) then
    print *, "read prt_function"; call parse_node_write (pn)
  end if
  pn_clause => parse_node_get_sub_ptr (pn)
  pn_key   => parse_node_get_sub_ptr (pn_clause)
  pn_cond => parse_node_get_next_ptr (pn_key)
  if (associated (pn_cond)) &
    pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
  pn_args => parse_node_get_next_ptr (pn_clause)
  pn_arg1 => parse_node_get_sub_ptr (pn_args)
  pn_arg2 => parse_node_get_next_ptr (pn_arg1)
  key = parse_node_get_key (pn_key)
  call eval_node_compile_pexpr (en1, pn_arg1, var_list)
  allocate (en)
  if (.not. associated (pn_arg2)) then
    select case (char (key))
    case ("collect")
      call eval_node_init_prt_fun_unary (en, en1, key, collect_p)
    case ("select")
      call eval_node_init_prt_fun_unary (en, en1, key, select_p)
    case ("extract")
      call eval_node_init_prt_fun_unary (en, en1, key, extract_p)
    case ("sort")
      call eval_node_init_prt_fun_unary (en, en1, key, sort_p)
    case default
      call msg_bug (" Unary particle function '" // char (key) // '&
        "' undefined")
    end select
  else
    call eval_node_compile_pexpr (en2, pn_arg2, var_list)
    select case (char (key))
    case ("join")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, join_pp)
    case ("combine")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, combine_pp)
    case ("collect")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, collect_pp)
    case ("select")
```

```

        call eval_node_init_prt_fun_binary (en, en1, en2, key, select_pp)
case ("sort")
    call eval_node_init_prt_fun_binary (en, en1, en2, key, sort_pp)
case default
    call msg_bug (" Binary particle function '" // char (key) // &
        "' undefined")
end select
end if
if (associated (pn_cond)) then
    call eval_node_set_observables (en, var_list)
    select case (char (key))
    case ("extract", "sort")
        call eval_node_compile_expr (en0, pn_arg0, en%var_list)
    case default
        call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
    end select
    en%arg0 => en0
end if
if (debug) then
    call eval_node_write (en)
    print *, "done prt_function"
end if
end subroutine eval_node_compile_prt_function

```

The `eval` expression is similar, but here the expression `arg0` is mandatory, and the whole thing evaluates to a numeric value.

(*Expressions: procedures*) +=

```

recursive subroutine eval_node_compile_eval_function (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_key, pn_arg0, pn_args, pn_arg1, pn_arg2
    type(eval_node_t), pointer :: en0, en1, en2
    type(string_t) :: key
    if (debug) then
        print *, "read eval_function"; call parse_node_write (pn)
    end if
    pn_key => parse_node_get_sub_ptr (pn)
    pn_arg0 => parse_node_get_next_ptr (pn_key)
    pn_args => parse_node_get_next_ptr (pn_arg0)
    pn_arg1 => parse_node_get_sub_ptr (pn_args)
    pn_arg2 => parse_node_get_next_ptr (pn_arg1)
    key = parse_node_get_key (pn_key)
    call eval_node_compile_pexpr (en1, pn_arg1, var_list)
    allocate (en)
    if (.not. associated (pn_arg2)) then
        call eval_node_init_eval_fun_unary (en, en1, key)
    else
        call eval_node_compile_pexpr (en2, pn_arg2, var_list)
        call eval_node_init_eval_fun_binary (en, en1, en2, key)
    end if
    call eval_node_set_observables (en, var_list)
    call eval_node_compile_expr (en0, pn_arg0, en%var_list)

```

```

if (en0%result_type /= V_REAL) &
    call msg_fatal (" 'eval' function does not result in real value")
call eval_node_set_expr (en, en0)
if (debug) then
    call eval_node_write (en)
    print *, "done eval_function"
end if
end subroutine eval_node_compile_eval_function

```

Logical functions of subevents.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_log_function (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_key, pn_str
    type(parse_node_t), pointer :: pn_arg0, pn_args, pn_arg1, pn_arg2
    type(eval_node_t), pointer :: en0, en1, en2
    type(string_t) :: key
    if (debug) then
        print *, "read log_function"; call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("all_fun", "any_fun", "no_fun")
        pn_key => parse_node_get_sub_ptr (pn)
        pn_arg0 => parse_node_get_next_ptr (pn_key)
        pn_args => parse_node_get_next_ptr (pn_arg0)
    case ("user_cut_fun")
        pn_key => parse_node_get_sub_ptr (pn)
        pn_str => parse_node_get_next_ptr (pn_key)
        pn_arg0 => parse_node_get_sub_ptr (pn_str)
        pn_args => parse_node_get_next_ptr (pn_str)
    case default
        call parse_node_mismatch &
            ("all_fun|any_fun|no_fun|user_cut_fun", &
            pn)
    end select
    pn_arg1 => parse_node_get_sub_ptr (pn_args)
    pn_arg2 => parse_node_get_next_ptr (pn_arg1)
    key = parse_node_get_key (pn_key)
    call eval_node_compile_pexpr (en1, pn_arg1, var_list)
    allocate (en)
    if (.not. associated (pn_arg2)) then
        select case (char (key))
        case ("all")
            call eval_node_init_log_fun_unary (en, en1, key, all_p)
        case ("any")
            call eval_node_init_log_fun_unary (en, en1, key, any_p)
        case ("no")
            call eval_node_init_log_fun_unary (en, en1, key, no_p)
        case ("user_cut")
            call eval_node_init_log_fun_unary (en, en1, key, user_cut_p)
        case default
            call msg_bug ("Unary logical particle function '" // char (key) // &

```

```

        "' undefined")
    end select
else
    call eval_node_compile_pexpr (en2, pn_arg2, var_list)
    select case (char (key))
    case ("all")
        call eval_node_init_log_fun_binary (en, en1, en2, key, all_pp)
    case ("any")
        call eval_node_init_log_fun_binary (en, en1, en2, key, any_pp)
    case ("no")
        call eval_node_init_log_fun_binary (en, en1, en2, key, no_pp)
    case default
        call msg_bug ("Binary logical particle function '" // char (key) // &
            "' undefined")
    end select
end if
if (associated (pn_arg0)) then
    call eval_node_set_observables (en, var_list)
    select case (char (key))
    case ("all", "any", "no")
        call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
    case ("user_cut")
        call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
    case default
        call msg_bug ("Compiling logical particle function: missing mode")
    end select
    call eval_node_set_expr (en, en0, V_LOG)
end if
if (debug) then
    call eval_node_write (en)
    print *, "done log_function"
end if
end subroutine eval_node_compile_log_function

```

Numeric functions of subevents.

*(Expressions: procedures)* +=

```

recursive subroutine eval_node_compile_numeric_function (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
    type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
    type(eval_node_t), pointer :: en0, en1, en2
    type(string_t) :: key
    if (debug) then
        print *, "read numeric_function"; call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("count_fun")
        pn_clause => parse_node_get_sub_ptr (pn)
        pn_key => parse_node_get_sub_ptr (pn_clause)
        pn_cond => parse_node_get_next_ptr (pn_key)
        if (associated (pn_cond)) then
            pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)

```

```

else
    pn_arg0 => null ()
end if
pn_args => parse_node_get_next_ptr (pn_clause)
case ("user_event_fun")
    pn_key => parse_node_get_sub_ptr (pn)
    pn_cond => parse_node_get_next_ptr (pn_key)
    pn_arg0 => parse_node_get_sub_ptr (pn_cond)
    pn_args => parse_node_get_next_ptr (pn_cond)
end select
pn_arg1 => parse_node_get_sub_ptr (pn_args)
pn_arg2 => parse_node_get_next_ptr (pn_arg1)
key = parse_node_get_key (pn_key)
call eval_node_compile_pexpr (en1, pn_arg1, var_list)
allocate (en)
if (.not. associated (pn_arg2)) then
    select case (char (key))
    case ("count")
        call eval_node_init_int_fun_unary (en, en1, key, count_a)
    case ("user_event_shape")
        call eval_node_init_real_fun_unary (en, en1, key, user_event_shape_a)
    case default
        call msg_bug ("Unary subevent function '" // char (key) // &
            "' undefined")
    end select
else
    call eval_node_compile_pexpr (en2, pn_arg2, var_list)
    select case (char (key))
    case ("count")
        call eval_node_init_int_fun_binary (en, en1, en2, key, count_pp)
    case default
        call msg_bug ("Binary subevent function '" // char (key) // &
            "' undefined")
    end select
end if
if (associated (pn_arg0)) then
    call eval_node_set_observables (en, var_list)
    select case (char (key))
    case ("count")
        call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
        call eval_node_set_expr (en, en0, V_INT)
    case ("user_event_shape")
        call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
        call eval_node_set_expr (en, en0, V_REAL)
    end select
end if
if (debug) then
    call eval_node_write (en)
    print *, "done numeric_function"
end if
end subroutine eval_node_compile_numeric_function

```



## PDG-code arrays

A PDG-code expression is either prefixed by `incoming` or `outgoing`, a block, or a conditional. In any case, it evaluates to a constant.

*(Expressions: procedures)*+≡

```
recursive subroutine eval_node_compile_prefix_cexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_avalue, pn_prt
  type(string_t) :: key
  if (debug) then
    print *, "read prefix_cexpr"; call parse_node_write (pn)
  end if
  pn_avalue => parse_node_get_sub_ptr (pn)
  key = parse_node_get_rule_key (pn_avalue)
  select case (char (key))
  case ("incoming_prt")
    pn_prt => parse_node_get_sub_ptr (pn_avalue, 2)
    call eval_node_compile_cexpr (en, pn_prt, var_list)
  case ("outgoing_prt")
    pn_prt => parse_node_get_sub_ptr (pn_avalue, 1)
    call eval_node_compile_cexpr (en, pn_prt, var_list)
  case default
    call parse_node_mismatch &
      ("incoming_prt|outgoing_prt", &
       pn_avalue)
  end select
  if (debug) then
    call eval_node_write (en)
    print *, "done prefix_cexpr"
  end if
end subroutine eval_node_compile_prefix_cexpr
```

A PDG array is a string of PDG code definitions (or aliases), concatenated by `','`. The code definitions may be variables which are not defined at compile time, so we have to allocate sub-nodes. This analogous to `eval_node_compile_term`.

*(Expressions: procedures)*+≡

```
recursive subroutine eval_node_compile_cexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_prt, pn_concatenation
  type(eval_node_t), pointer :: en1, en2
  type(pdg_array_t) :: aval
  if (debug) then
    print *, "read cexpr"; call parse_node_write (pn)
  end if
  pn_prt => parse_node_get_sub_ptr (pn)
  call eval_node_compile_avalue (en, pn_prt, var_list)
  pn_concatenation => parse_node_get_next_ptr (pn_prt)
  do while (associated (pn_concatenation))
    pn_prt => parse_node_get_sub_ptr (pn_concatenation, 2)
```

```

en1 => en
call eval_node_compile_avalue (en2, pn_prt, var_list)
allocate (en)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
    call concat_cc (aval, en1, en2)
    call eval_node_init_pdg_array (en, aval)
    call eval_node_final_rec (en1)
    call eval_node_final_rec (en2)
    deallocate (en1, en2)
else
    call eval_node_init_branch (en, var_str (":"), V_PDG, en1, en2)
    call eval_node_set_op2_pdg (en, concat_cc)
end if
pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
end do
if (debug) then
    call eval_node_write (en)
    print *, "done cexpr"
end if
end subroutine eval_node_compile_cexpr

```

Compile a PDG-code type value. It may be either an integer expression or a variable of type PDG array, optionally quoted.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_avalue (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    if (debug) then
        print *, "read avalue"; call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("pdg_code")
        call eval_node_compile_pdg_code (en, pn, var_list)
    case ("cvariable", "variable", "prt_name")
        call eval_node_compile_cvariable (en, pn, var_list)
    case ("cexpr")
        call eval_node_compile_cexpr (en, pn, var_list)
    case ("block_cexpr")
        call eval_node_compile_block_expr (en, pn, var_list, V_PDG)
    case ("conditional_cexpr")
        call eval_node_compile_conditional (en, pn, var_list, V_PDG)
    case default
        call parse_node_mismatch &
            ("grouped_cexpr|block_cexpr|conditional_cexpr|" // &
             "pdg_code|cvariable|prt_name", pn)
    end select
    if (debug) then
        call eval_node_write (en)
        print *, "done avalue"
    end if
end subroutine eval_node_compile_avalue

```

Compile a PDG-code expression, which is the key PDG with an integer expression as argument. The procedure is analogous to `eval_node_compile_unary_function`.

*(Expressions: procedures)* +≡

```

subroutine eval_node_compile_pdg_code (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_arg
  type(eval_node_t), pointer :: en1
  type(string_t) :: key
  type(pdg_array_t) :: aval
  integer :: t
  if (debug) then
    print *, "read PDG code"; call parse_node_write (pn)
  end if
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  call eval_node_compile_expr &
    (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
  t = en1%result_type
  allocate (en)
  key = "PDG"
  if (en1%type == EN_CONSTANT) then
    select case (t)
    case (V_INT)
      call pdg_i (aval, en1)
      call eval_node_init_pdg_array (en, aval)
    case default; call eval_type_error (pn, char (key), t)
    end select
    call eval_node_final_rec (en1)
    deallocate (en1)
  else
    select case (t)
    case (V_INT); call eval_node_set_op1_pdg (en, pdg_i)
    case default; call eval_type_error (pn, char (key), t)
    end select
  end if
  if (debug) then
    call eval_node_write (en)
    print *, "done function"
  end if
end subroutine eval_node_compile_pdg_code

```

This is entirely analogous to `eval_node_compile_variable`. However, PDG-array variables occur in different contexts.

To avoid name clashes between PDG-array variables and ordinary variables, we prepend a character (\*). This is not visible to the user.

*(Expressions: procedures)* +≡

```

subroutine eval_node_compile_cvariable (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_name
  type(string_t) :: var_name

```

```

type(var_entry_t), pointer :: var
type(pdg_array_t), target, save :: no_aval
logical, target, save :: unknown = .false.
if (debug) then
    print *, "read cvariable"; call parse_node_write (pn)
end if
!   select case (char (parse_node_get_rule_key (pn)))
!   case ("cvariable"); pn_name => parse_node_get_sub_ptr (pn, 2)
!   case default;      pn_name => pn
!   end select
pn_name => pn
var_name = parse_node_get_string (pn_name)
var => var_list_get_var_ptr (var_list, var_name, V_PDG, defined=.true.)
allocate (en)
if (associated (var)) then
    call eval_node_init_pdg_array_ptr &
        (en, var_entry_get_name (var), var_entry_get_aval_ptr (var), &
        var_entry_get_known_ptr (var))
else
    call parse_node_write (pn)
    call msg_error ("This PDG-array variable is undefined at this point")
    call eval_node_init_pdg_array_ptr (en, var_name, no_aval, unknown)
end if
if (debug) then
    call eval_node_write (en)
    print *, "done cvariable"
end if
end subroutine eval_node_compile_cvariable

```

## String expressions

A string expression is either a string value or a concatenation of string values.

*(Expressions: procedures)* +≡

```

recursive subroutine eval_node_compile_sexpr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_svalue, pn_concatenation, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: string
    if (debug) then
        print *, "read sexpr"; call parse_node_write (pn)
    end if
    pn_svalue => parse_node_get_sub_ptr (pn)
    call eval_node_compile_svalue (en, pn_svalue, var_list)
    pn_concatenation => &
        parse_node_get_next_ptr (pn_svalue, tag="str_concatenation")
    do while (associated (pn_concatenation))
        pn_op => parse_node_get_sub_ptr (pn_concatenation)
        pn_arg => parse_node_get_next_ptr (pn_op)
        en1 => en
        call eval_node_compile_svalue (en2, pn_arg, var_list)
        allocate (en)
    end do

```

```

        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call concat_ss (string, en1, en2)
            call eval_node_init_string (en, string)
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
        else
            call eval_node_init_branch &
                (en, var_str ("concat"), V_STR, en1, en2)
            call eval_node_set_op2_str (en, concat_ss)
        end if
        pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
    end do
    if (debug) then
        call eval_node_write (en)
        print *, "done sexpr"
    end if
end subroutine eval_node_compile_sexpr

```

A string value is a string literal, a variable, a (grouped) sexpr, a block sexpr, or a conditional.

*(Expressions: procedures)* +=

```

recursive subroutine eval_node_compile_svalue (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    if (debug) then
        print *, "read svalue"; call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("svariable")
        call eval_node_compile_variable (en, pn, var_list, V_STR)
    case ("sexpr")
        call eval_node_compile_sexpr (en, pn, var_list)
    case ("block_sexpr")
        call eval_node_compile_block_expr (en, pn, var_list, V_STR)
    case ("conditional_sexpr")
        call eval_node_compile_conditional (en, pn, var_list, V_STR)
    case ("sprintf_fun")
        call eval_node_compile_sprintf (en, pn, var_list)
    case ("sprintfd_fun")
        call eval_node_compile_sprintfd (en, pn, var_list)
    case ("string_literal")
        allocate (en)
        call eval_node_init_string (en, parse_node_get_string (pn))
    case default
        call parse_node_mismatch &
            ("svariable|" // &
            "grouped_sexpr|block_sexpr|conditional_sexpr|" // &
            "string_function|string_literal", pn)
    end select
    if (debug) then
        call eval_node_write (en)
    end if
end subroutine eval_node_compile_svalue

```

```

        print *, "done svalue"
    end if
end subroutine eval_node_compile_svalue

```

There are currently one string function, `sprintf` and `sprintfd`. For `sprintf`, The first argument (no brackets) is the format string, the optional arguments in brackets are the expressions or variables to be formatted.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_sprintf (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_clause, pn_key, pn_args
    type(parse_node_t), pointer :: pn_arg0
    type(eval_node_t), pointer :: en0, en1
    integer :: n_args
    type(string_t) :: key
    if (debug) then
        print *, "read sprintf_fun"; call parse_node_write (pn)
    end if
    pn_clause => parse_node_get_sub_ptr (pn)
    pn_key => parse_node_get_sub_ptr (pn_clause)
    pn_arg0 => parse_node_get_next_ptr (pn_key)
    pn_args => parse_node_get_next_ptr (pn_clause)
    call eval_node_compile_sexpr (en0, pn_arg0, var_list)
    if (associated (pn_args)) then
        call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
    else
        n_args = 0
        en1 => null ()
    end if
    allocate (en)
    key = parse_node_get_key (pn_key)
    call eval_node_init_format_string (en, en0, en1, key, n_args)
    if (debug) then
        call eval_node_write (en)
        print *, "done sprintf_fun"
    end if
end subroutine eval_node_compile_sprintf

```

`sprintfd` is similar, just the format string is missing.

*(Expressions: procedures)*+≡

```

recursive subroutine eval_node_compile_sprintfd (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_key, pn_args
    type(eval_node_t), pointer :: en1
    integer :: n_args
    type(string_t) :: key
    if (debug) then
        print *, "read sprintfd_fun"; call parse_node_write (pn)
    end if

```

```

pn_key => parse_node_get_sub_ptr (pn)
pn_args => parse_node_get_next_ptr (pn_key)
if (associated (pn_args)) then
    call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
else
    n_args = 0
    en1 => null ()
end if
allocate (en)
key = parse_node_get_key (pn_key)
call eval_node_init_format_string (en, null (), en1, key, n_args)
if (debug) then
    call eval_node_write (en)
    print *, "done sprintf_fun"
end if
end subroutine eval_node_compile_sprintfd

```

*(Expressions: procedures)*+≡

```

subroutine eval_node_compile_sprintf_args (en, pn, var_list, n_args)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(out) :: n_args
    type(parse_node_t), pointer :: pn_arg
    integer :: i
    type(eval_node_t), pointer :: en1, en2
    n_args = parse_node_get_n_sub (pn)
    en => null ()
    do i = n_args, 1, -1
        pn_arg => parse_node_get_sub_ptr (pn, i)
        select case (char (parse_node_get_rule_key (pn_arg)))
            case ("lvariable")
                call eval_node_compile_variable (en1, pn_arg, var_list, V_LOG)
            case ("svariable")
                call eval_node_compile_variable (en1, pn_arg, var_list, V_STR)
            case ("expr")
                call eval_node_compile_expr (en1, pn_arg, var_list)
            case default
                call parse_node_mismatch ("variable|svariable|lvariable|expr", pn_arg)
        end select
        if (associated (en)) then
            en2 => en
            allocate (en)
            call eval_node_init_branch &
                (en, var_str ("sprintf_arg"), V_NONE, en1, en2)
        else
            allocate (en)
            call eval_node_init_branch &
                (en, var_str ("sprintf_arg"), V_NONE, en1)
        end if
    end do
end subroutine eval_node_compile_sprintf_args

```

Evaluation. We allocate the argument list and apply the Fortran wrapper for the `sprintf` function. In the `sprintfd` case, build a format string appropriate for the argument types.

*(Expressions: procedures)*+≡

```

subroutine evaluate_sprintf (string, n_args, en_fmt, en_arg)
  type(string_t), intent(out) :: string
  integer, intent(in) :: n_args
  type(eval_node_t), pointer :: en_fmt
  type(eval_node_t), intent(in), optional, target :: en_arg
  type(eval_node_t), pointer :: en_branch, en_var
  type(sprintf_arg_t), dimension(:), allocatable :: arg
  type(string_t) :: fmt
  logical :: autoformat
  integer :: i, j, sprintf_argc
  autoformat = .not. associated (en_fmt)
  if (autoformat) fmt = ""
  if (present (en_arg)) then
    sprintf_argc = 0
    en_branch => en_arg
    do i = 1, n_args
      select case (en_branch%arg1%result_type)
        case (V_CMPLX); sprintf_argc = sprintf_argc + 2
        case default ; sprintf_argc = sprintf_argc + 1
      end select
      en_branch => en_branch%arg2
    end do
    allocate (arg (sprintf_argc))
    j = 1
    en_branch => en_arg
    do i = 1, n_args
      en_var => en_branch%arg1
      select case (en_var%result_type)
        case (V_LOG)
          call sprintf_arg_init (arg(j), en_var%lval)
          if (autoformat) fmt = fmt // "%s "
        case (V_INT);
          call sprintf_arg_init (arg(j), en_var%ival)
          if (autoformat) fmt = fmt // "%i "
        case (V_REAL);
          call sprintf_arg_init (arg(j), en_var%rval)
          if (autoformat) fmt = fmt // "%g "
        case (V_STR)
          call sprintf_arg_init (arg(j), en_var%sval)
          if (autoformat) fmt = fmt // "%s "
        case (V_CMPLX)
          call sprintf_arg_init (arg(j), real (en_var%cval, default))
          j = j + 1
          call sprintf_arg_init (arg(j), aimag (en_var%cval))
          if (autoformat) fmt = fmt // "(%g + %g * I) "
        case default
          call eval_node_write (en_var)
          call msg_error ("sprintf is implemented " &
            // "for logical, integer, real, and string values only")
      end select
    end do
  end if
end subroutine evaluate_sprintf

```



```

        j = j + 1
        en_branch => en_branch%arg2
    end do
else
    allocate (arg(0))
end if
if (autoformat) then
    string = sprintf (trim (fmt), arg)
else
    string = sprintf (en_fmt%sval, arg)
end if
end subroutine evaluate_sprintf

```

### 5.6.5 Auxiliary functions for the compiler

Issue an error that the current node could not be compiled because of type mismatch:

```

(Expressions: procedures) +=
subroutine eval_type_error (pn, string, t)
    type(parse_node_t), intent(in) :: pn
    character(*), intent(in) :: string
    integer, intent(in) :: t
    type(string_t) :: type
    select case (t)
    case (V_NONE); type = "(none)"
    case (V_LOG); type = "'logical'"
    case (V_INT); type = "'integer'"
    case (V_REAL); type = "'real'"
    case (V_CMPLX); type = "'complex'"
    case default; type = "(unknown)"
    end select
    call parse_node_write (pn)
    call msg_fatal (" The " // string // &
        " operation is not defined for the given argument type " // &
        char (type))
end subroutine eval_type_error

```

If two numerics are combined, the result is integer if both arguments are integer, if one is integer and the other real or both are real, than its argument is real, otherwise complex.

```

(Expressions: procedures) +=
function numeric_result_type (t1, t2) result (t)
    integer, intent(in) :: t1, t2
    integer :: t
    if (t1 == V_INT .and. t2 == V_INT) then
        t = V_INT
    else if (t1 == V_INT .and. t2 == V_REAL) then
        t = V_REAL
    else if (t1 == V_REAL .and. t2 == V_INT) then
        t = V_REAL
    else if (t1 == V_REAL .and. t2 == V_REAL) then
        t = V_REAL
    end if
end function

```

```

else
    t = V_CMPLX
end if
end function numeric_result_type

```

### 5.6.6 Evaluation

Evaluation is done recursively. For leaf nodes nothing is to be done.

Evaluating particle-list functions: First, we evaluate the particle lists. If a condition is present, we assign the particle pointers of the condition node to the allocated particle entries in the parent node, keeping in mind that the observables in the variable stack used for the evaluation of the condition also contain pointers to these entries. Then, the assigned procedure is evaluated, which sets the subevent in the parent node. If required, the procedure evaluates the condition node once for each (pair of) particles to determine the result.

*(Expressions: procedures)+≡*

```

recursive subroutine eval_node_evaluate (en)
    type(eval_node_t), intent(inout) :: en
    logical :: exist
    select case (en%type)
    case (EN_UNARY)
        if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)
            en%value_is_known = en%arg1%value_is_known
        else
            en%value_is_known = .false.
        end if
        if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG); en%lval = en% op1_log (en%arg1)
            case (V_INT); en%ival = en% op1_int (en%arg1)
            case (V_REAL); en%rval = en% op1_real (en%arg1)
            case (V_CMPLX); en%cval = en% op1_cmplx (en%arg1)
            case (V_PDG);
                call en% op1_pdg (en%aval, en%arg1)
            case (V_SEV)
                if (associated (en%arg0)) then
                    call en% op1_sev (en%pval, en%arg1, en%arg0)
                else
                    call en% op1_sev (en%pval, en%arg1)
                end if
            case (V_STR)
                call en% op1_str (en%sval, en%arg1)
            end select
        end if
    case (EN_BINARY)
        if (associated (en%arg1) .and. associated (en%arg2)) then
            call eval_node_evaluate (en%arg1)
            call eval_node_evaluate (en%arg2)
            en%value_is_known = &
                en%arg1%value_is_known .and. en%arg2%value_is_known
        else

```

```

        en%value_is_known = .false.
    end if
    if (en%value_is_known) then
        select case (en%result_type)
        case (V_LOG); en%lval = en% op2_log (en%arg1, en%arg2)
        case (V_INT); en%ival = en% op2_int (en%arg1, en%arg2)
        case (V_REAL); en%rval = en% op2_real (en%arg1, en%arg2)
        case (V_CMPLX); en%cval = en% op2_cmplx (en%arg1, en%arg2)
        case (V_PDG)
            call en% op2_pdg (en%aval, en%arg1, en%arg2)
        case (V_SEV)
            if (associated (en%arg0)) then
                call en% op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
            else
                call en% op2_sev (en%pval, en%arg1, en%arg2)
            end if
        case (V_STR)
            call en% op2_str (en%sval, en%arg1, en%arg2)
        end select
    end if
case (EN_BLOCK)
    if (associated (en%arg1) .and. associated (en%arg0)) then
        call eval_node_evaluate (en%arg1)
        call eval_node_evaluate (en%arg0)
        en%value_is_known = en%arg0%value_is_known
    else
        en%value_is_known = .false.
    end if
    if (en%value_is_known) then
        select case (en%result_type)
        case (V_LOG); en%lval = en%arg0%lval
        case (V_INT); en%ival = en%arg0%ival
        case (V_REAL); en%rval = en%arg0%rval
        case (V_CMPLX); en%cval = en%arg0%cval
        case (V_PDG); en%aval = en%arg0%aval
        case (V_SEV); en%pval = en%arg0%pval
        case (V_STR); en%sval = en%arg0%sval
        end select
    end if
case (EN_CONDITIONAL)
    if (associated (en%arg0)) then
        call eval_node_evaluate (en%arg0)
        en%value_is_known = en%arg0%value_is_known
    else
        en%value_is_known = .false.
    end if
    if (en%arg0%value_is_known) then
        if (en%arg0%lval) then
            call eval_node_evaluate (en%arg1)
            en%value_is_known = en%arg1%value_is_known
            if (en%value_is_known) then
                select case (en%result_type)
                case (V_LOG); en%lval = en%arg1%lval
                case (V_INT); en%ival = en%arg1%ival

```

```

        case (V_REAL); en%rval = en%arg1%rval
        case (V_CMPLX); en%cval = en%arg1%cval
        case (V_PDG);  en%aval = en%arg1%aval
        case (V_SEV);  en%pval = en%arg1%pval
        case (V_STR);  en%sval = en%arg1%sval
    end select
end if
else
    call eval_node_evaluate (en%arg2)
    en%value_is_known = en%arg2%value_is_known
    if (en%value_is_known) then
        select case (en%result_type)
        case (V_LOG);  en%lval = en%arg2%lval
        case (V_INT);  en%ival = en%arg2%ival
        case (V_REAL); en%rval = en%arg2%rval
        case (V_CMPLX); en%cval = en%arg2%cval
        case (V_PDG);  en%aval = en%arg2%aval
        case (V_SEV);  en%pval = en%arg2%pval
        case (V_STR);  en%sval = en%arg2%sval
        end select
    end if
end if
end if
!    call eval_node_write_rec (en)
case (EN_RECORD_CMD)
    exist = .true.
    en%lval = .false.
    call eval_node_evaluate (en%arg0)
    if (en%arg0%value_is_known) then
        if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)
            if (en%arg1%value_is_known) then
                if (associated (en%arg2)) then
                    call eval_node_evaluate (en%arg2)
                    if (en%arg2%value_is_known) then
                        if (associated (en%arg3)) then
                            call eval_node_evaluate (en%arg3)
                            if (en%arg3%value_is_known) then
                                if (associated (en%arg4)) then
                                    call eval_node_evaluate (en%arg4)
                                    if (en%arg4%value_is_known) then
                                        if (associated (en%rval)) then
                                            call analysis_record_data (en%arg0%sval, &
                                                en%arg1%rval, en%arg2%rval, &
                                                en%arg3%rval, en%arg4%rval, &
                                                weight=en%rval, exist=exist, &
                                                success=en%lval)
                                        else
                                            call analysis_record_data (en%arg0%sval, &
                                                en%arg1%rval, en%arg2%rval, &
                                                en%arg3%rval, en%arg4%rval, &
                                                exist=exist, success=en%lval)
                                        end if
                                    end if
                                end if
                            end if
                        end if
                    end if
                end if
            end if
        end if
    end if
end if

```

```

        else
            if (associated (en%rval)) then
                call analysis_record_data (en%arg0%sval, &
                    en%arg1%rval, en%arg2%rval, &
                    en%arg3%rval, &
                    weight=en%rval, exist=exist, &
                    success=en%lval)
            else
                call analysis_record_data (en%arg0%sval, &
                    en%arg1%rval, en%arg2%rval, &
                    en%arg3%rval, &
                    exist=exist, success=en%lval)
            end if
        end if
    end if
else
    if (associated (en%rval)) then
        call analysis_record_data (en%arg0%sval, &
            en%arg1%rval, en%arg2%rval, &
            weight=en%rval, exist=exist, &
            success=en%lval)
    else
        call analysis_record_data (en%arg0%sval, &
            en%arg1%rval, en%arg2%rval, &
            exist=exist, success=en%lval)
    end if
end if
end if
else
    if (associated (en%rval)) then
        call analysis_record_data (en%arg0%sval, &
            en%arg1%rval, &
            weight=en%rval, exist=exist, success=en%lval)
    else
        call analysis_record_data (en%arg0%sval, &
            en%arg1%rval, &
            exist=exist, success=en%lval)
    end if
end if
end if
else
    if (associated (en%rval)) then
        call analysis_record_data (en%arg0%sval, 1._default, &
            weight=en%rval, exist=exist, success=en%lval)
    else
        call analysis_record_data (en%arg0%sval, 1._default, &
            exist=exist, success=en%lval)
    end if
end if
if (.not. exist) then
    call msg_error ("Analysis object '" // char (en%arg0%sval) &
        // "' is undefined")
    en%arg0%value_is_known = .false.
end if

```

```

        end if
    case (EN_OBS1_INT)
        en%ival = en% obs1_int (en%prt1)
        en%value_is_known = .true.
    case (EN_OBS2_INT)
        en%ival = en% obs2_int (en%prt1, en%prt2)
        en%value_is_known = .true.
    case (EN_OBS1_REAL)
        en%rval = en% obs1_real (en%prt1)
        en%value_is_known = .true.
    case (EN_OBS2_REAL)
        en%rval = en% obs2_real (en%prt1, en%prt2)
        en%value_is_known = .true.
    case (EN_UOBS1_INT)
        en%ival = user_obs_int_p (en%arg0, en%prt1)
        en%value_is_known = .true.
    case (EN_UOBS2_INT)
        en%ival = user_obs_int_pp (en%arg0, en%prt1, en%prt2)
        en%value_is_known = .true.
    case (EN_UOBS1_REAL)
        en%rval = user_obs_real_p (en%arg0, en%prt1)
        en%value_is_known = .true.
    case (EN_UOBS2_REAL)
        en%rval = user_obs_real_pp (en%arg0, en%prt1, en%prt2)
        en%value_is_known = .true.
    case (EN_PRT_FUN_UNARY)
        call eval_node_evaluate (en%arg1)
        en%value_is_known = en%arg1%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                call en% op1_sev (en%pval, en%arg1, en%arg0)
            else
                call en% op1_sev (en%pval, en%arg1)
            end if
        end if
    case (EN_PRT_FUN_BINARY)
        call eval_node_evaluate (en%arg1)
        call eval_node_evaluate (en%arg2)
        en%value_is_known = &
            en%arg1%value_is_known .and. en%arg2%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                en%arg0%prt2 => en%prt2
                call en% op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
            else
                call en% op2_sev (en%pval, en%arg1, en%arg2)
            end if
        end if
    case (EN_EVAL_FUN_UNARY)
        call eval_node_evaluate (en%arg1)

```

```

    en%value_is_known = subevt_is_nonempty (en%arg1%pval)
    if (en%value_is_known) then
        en%arg0%index => en%index
        en%index = 1
        en%arg0%prt1 => en%prt1
        en%prt1 = subevt_get_prt (en%arg1%pval, 1)
        call eval_node_evaluate (en%arg0)
        en%rval = en%arg0%rval
    end if
case (EN_EVAL_FUN_BINARY)
    call eval_node_evaluate (en%arg1)
    call eval_node_evaluate (en%arg2)
    en%value_is_known = &
        subevt_is_nonempty (en%arg1%pval) .and. &
        subevt_is_nonempty (en%arg2%pval)
    if (en%value_is_known) then
        en%arg0%index => en%index
        en%arg0%prt1 => en%prt1
        en%arg0%prt2 => en%prt2
        en%index = 1
        en%prt1 = subevt_get_prt (en%arg1%pval, 1)
        en%prt2 = subevt_get_prt (en%arg2%pval, 1)
        call eval_node_evaluate (en%arg0)
        en%rval = en%arg0%rval
    end if
case (EN_LOG_FUN_UNARY)
    call eval_node_evaluate (en%arg1)
    en%value_is_known = .true.
    if (en%value_is_known) then
        en%arg0%index => en%index
        en%arg0%prt1 => en%prt1
        en%lval = en% op1_cut (en%arg1, en%arg0)
    end if
case (EN_LOG_FUN_BINARY)
    call eval_node_evaluate (en%arg1)
    call eval_node_evaluate (en%arg2)
    en%value_is_known = .true.
    if (en%value_is_known) then
        en%arg0%index => en%index
        en%arg0%prt1 => en%prt1
        en%arg0%prt2 => en%prt2
        en%lval = en% op2_cut (en%arg1, en%arg2, en%arg0)
    end if
case (EN_INT_FUN_UNARY)
    call eval_node_evaluate (en%arg1)
    en%value_is_known = en%arg1%value_is_known
    if (en%value_is_known) then
        if (associated (en%arg0)) then
            en%arg0%index => en%index
            en%arg0%prt1 => en%prt1
            call en% op1_evi (en%ival, en%arg1, en%arg0)
        else
            call en% op1_evi (en%ival, en%arg1)
        end if
    end if

```

```

        end if
    case (EN_INT_FUN_BINARY)
        call eval_node_evaluate (en%arg1)
        call eval_node_evaluate (en%arg2)
        en%value_is_known = &
            en%arg1%value_is_known .and. &
            en%arg2%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                en%arg0%prt2 => en%prt2
                call en% op2_evi (en%ival, en%arg1, en%arg2, en%arg0)
            else
                call en% op2_evi (en%ival, en%arg1, en%arg2)
            end if
        end if
    case (EN_REAL_FUN_UNARY)
        call eval_node_evaluate (en%arg1)
        en%value_is_known = en%arg1%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                call en% op1_evr (en%rval, en%arg1, en%arg0)
            else
                call en% op1_evr (en%rval, en%arg1)
            end if
        end if
    case (EN_REAL_FUN_BINARY)
        call eval_node_evaluate (en%arg1)
        call eval_node_evaluate (en%arg2)
        en%value_is_known = &
            en%arg1%value_is_known .and. &
            en%arg2%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                en%arg0%prt2 => en%prt2
                call en% op2_evr (en%rval, en%arg1, en%arg2, en%arg0)
            else
                call en% op2_evr (en%rval, en%arg1, en%arg2)
            end if
        end if
    case (EN_FORMAT_STR)
        if (associated (en%arg0)) then
            call eval_node_evaluate (en%arg0)
            en%value_is_known = en%arg0%value_is_known
        else
            en%value_is_known = .true.
        end if
        if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)

```



```

        en%value_is_known = &
            en%value_is_known .and. en%arg1%value_is_known
        if (en%value_is_known) then
            call evaluate_sprintf (en%sval, en%ival, en%arg0, en%arg1)
        end if
    else
        if (en%value_is_known) then
            call evaluate_sprintf (en%sval, en%ival, en%arg0)
        end if
    end if
end select
if (debug) then
    print *, "evaluated"
    call eval_node_write (en)
end if
end subroutine eval_node_evaluate

```

### 5.6.7 Evaluation syntax

We have two different flavors of the syntax: with and without particles.

*(Expressions: variables)*+≡  
 type(syntax\_t), target, save :: syntax\_expr  
 type(syntax\_t), target, save :: syntax\_pexpr

These are for testing only and may be removed:

*(Expressions: public)*≡  
 public :: syntax\_expr\_init  
 public :: syntax\_pexpr\_init

*(Expressions: procedures)*+≡  
 subroutine syntax\_expr\_init ()  
 type(ifile\_t) :: ifile  
 call define\_expr\_syntax (ifile, particles=.false., analysis=.false.)  
 call syntax\_init (syntax\_expr, ifile)  
 call ifile\_final (ifile)  
 end subroutine syntax\_expr\_init

subroutine syntax\_pexpr\_init ()  
 type(ifile\_t) :: ifile  
 call define\_expr\_syntax (ifile, particles=.true., analysis=.false.)  
 call syntax\_init (syntax\_pexpr, ifile)  
 call ifile\_final (ifile)  
end subroutine syntax\_pexpr\_init

*(Expressions: public)*+≡  
 public :: syntax\_expr\_final  
 public :: syntax\_pexpr\_final

*(Expressions: procedures)*+≡  
 subroutine syntax\_expr\_final ()  
 call syntax\_final (syntax\_expr)  
 end subroutine syntax\_expr\_final

```

subroutine syntax_pexpr_final ()
  call syntax_final (syntax_pexpr)
end subroutine syntax_pexpr_final

```

*<Expressions: public>+≡*

```

public :: syntax_pexpr_write

```

*<Expressions: procedures>+≡*

```

subroutine syntax_pexpr_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_pexpr, unit)
end subroutine syntax_pexpr_write

```

*<Expressions: public>+≡*

```

public :: define_expr_syntax

```

Numeric expressions.

*<Expressions: procedures>+≡*

```

subroutine define_expr_syntax (ifile, particles, analysis)
  type(ifile_t), intent(inout) :: ifile
  logical, intent(in) :: particles, analysis
  type(string_t) :: numeric_pexpr
  type(string_t) :: var_plist, var_alias
  if (particles) then
    numeric_pexpr = " | numeric_pexpr"
    var_plist = " | var_plist"
    var_alias = " | var_alias"
  else
    numeric_pexpr = ""
    var_plist = ""
    var_alias = ""
  end if
  call ifile_append (ifile, "SEQ expr = subexpr addition*")
  call ifile_append (ifile, "ALT subexpr = addition | term")
  call ifile_append (ifile, "SEQ addition = plus_or_minus term")
  call ifile_append (ifile, "SEQ term = factor multiplication*")
  call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
  call ifile_append (ifile, "SEQ factor = value exponentiation?")
  call ifile_append (ifile, "SEQ exponentiation = to_the value")
  call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
  call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
  call ifile_append (ifile, "ALT to_the = '^' | '**")
  call ifile_append (ifile, "KEY '+'")
  call ifile_append (ifile, "KEY '-'")
  call ifile_append (ifile, "KEY '*'")
  call ifile_append (ifile, "KEY '/'")
  call ifile_append (ifile, "KEY '^'")
  call ifile_append (ifile, "KEY '**")
  call ifile_append (ifile, "ALT value = signed_value | unsigned_value")
  call ifile_append (ifile, "SEQ signed_value = '-' unsigned_value")
  call ifile_append (ifile, "ALT unsigned_value = " // &
    "numeric_value | constant | variable | " // &
    "result | user_observable | " // &
    "grouped_expr | block_expr | conditional_expr | " // &

```

```

        "unary_function | binary_function" // &
        numeric_pexpr)
call ifile_append (ifile, "ALT numeric_value = integer_value | " &
// "real_value | complex_value")
call ifile_append (ifile, "SEQ integer_value = integer_literal unit_expr?")
call ifile_append (ifile, "SEQ real_value = real_literal unit_expr?")
call ifile_append (ifile, "SEQ complex_value = complex_literal unit_expr?")
call ifile_append (ifile, "INT integer_literal")
call ifile_append (ifile, "REA real_literal")
call ifile_append (ifile, "COM complex_literal")
call ifile_append (ifile, "SEQ unit_expr = unit unit_power?")
call ifile_append (ifile, "ALT unit = " // &
"TeV | GeV | MeV | keV | eV | meV | " // &
"nbarn | pbarn | fbarn | abarn | " // &
"rad | mrad | degree | '%')")
call ifile_append (ifile, "KEY TeV")
call ifile_append (ifile, "KEY GeV")
call ifile_append (ifile, "KEY MeV")
call ifile_append (ifile, "KEY keV")
call ifile_append (ifile, "KEY eV")
call ifile_append (ifile, "KEY meV")
call ifile_append (ifile, "KEY nbarn")
call ifile_append (ifile, "KEY pbarn")
call ifile_append (ifile, "KEY fbarn")
call ifile_append (ifile, "KEY abarn")
call ifile_append (ifile, "KEY rad")
call ifile_append (ifile, "KEY mrad")
call ifile_append (ifile, "KEY degree")
call ifile_append (ifile, "KEY '%')")
call ifile_append (ifile, "SEQ unit_power = '^' frac_expr")
call ifile_append (ifile, "ALT frac_expr = frac | grouped_frac")
call ifile_append (ifile, "GRO grouped_frac = ( frac_expr )")
call ifile_append (ifile, "SEQ frac = signed_int div?")
call ifile_append (ifile, "ALT signed_int = " &
// "neg_int | pos_int | integer_literal")
call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
call ifile_append (ifile, "SEQ div = '/' integer_literal")
call ifile_append (ifile, "ALT constant = pi | I")
call ifile_append (ifile, "KEY pi")
call ifile_append (ifile, "KEY I")
call ifile_append (ifile, "IDE variable")
call ifile_append (ifile, "SEQ result = result_key result_arg")
call ifile_append (ifile, "ALT result_key = " // &
"num_id | n_calls | integral | error | accuracy | efficiency | chi2")
call ifile_append (ifile, "SEQ user_observable = user_obs user_arg")
call ifile_append (ifile, "KEY user_obs")
call ifile_append (ifile, "ARG user_arg = ( sexpr )")
call ifile_append (ifile, "KEY num_id")
call ifile_append (ifile, "KEY n_calls")
call ifile_append (ifile, "KEY integral")
call ifile_append (ifile, "KEY error")
call ifile_append (ifile, "KEY accuracy")
call ifile_append (ifile, "KEY efficiency")

```

```

call ifile_append (ifile, "KEY chi2")
call ifile_append (ifile, "GRO result_arg = ( process_id )")
call ifile_append (ifile, "IDE process_id")
call ifile_append (ifile, "SEQ unary_function = fun_unary function_arg1")
call ifile_append (ifile, "SEQ binary_function = fun_binary function_arg2")
call ifile_append (ifile, "ALT fun_unary = " // &
    "complex | real | int | nint | floor | ceiling | abs | sgn | " // &
    "sqrt | exp | log | log10 | " // &
    "sin | cos | tan | asin | acos | atan | " // &
    "sinh | cosh | tanh")
call ifile_append (ifile, "KEY complex")
call ifile_append (ifile, "KEY real")
call ifile_append (ifile, "KEY int")
call ifile_append (ifile, "KEY nint")
call ifile_append (ifile, "KEY floor")
call ifile_append (ifile, "KEY ceiling")
call ifile_append (ifile, "KEY abs")
call ifile_append (ifile, "KEY sgn")
call ifile_append (ifile, "KEY sqrt")
call ifile_append (ifile, "KEY exp")
call ifile_append (ifile, "KEY log")
call ifile_append (ifile, "KEY log10")
call ifile_append (ifile, "KEY sin")
call ifile_append (ifile, "KEY cos")
call ifile_append (ifile, "KEY tan")
call ifile_append (ifile, "KEY asin")
call ifile_append (ifile, "KEY acos")
call ifile_append (ifile, "KEY atan")
call ifile_append (ifile, "KEY sinh")
call ifile_append (ifile, "KEY cosh")
call ifile_append (ifile, "KEY tanh")
call ifile_append (ifile, "ALT fun_binary = max | min | mod | modulo")
call ifile_append (ifile, "KEY max")
call ifile_append (ifile, "KEY min")
call ifile_append (ifile, "KEY mod")
call ifile_append (ifile, "KEY modulo")
call ifile_append (ifile, "ARG function_arg1 = ( expr )")
call ifile_append (ifile, "ARG function_arg2 = ( expr, expr )")
call ifile_append (ifile, "GRO grouped_expr = ( expr )")
call ifile_append (ifile, "SEQ block_expr = let var_spec in expr")
call ifile_append (ifile, "KEY let")
call ifile_append (ifile, "ALT var_spec = " // &
    "var_num | var_int | var_real | var_complex | " // &
    "var_logical" // var_plist // var_alias // " | var_string")
call ifile_append (ifile, "SEQ var_num = var_name '=' expr")
call ifile_append (ifile, "SEQ var_int = int var_name '=' expr")
call ifile_append (ifile, "SEQ var_real = real var_name '=' expr")
call ifile_append (ifile, "SEQ var_complex = complex var_name '=' complex_expr")
call ifile_append (ifile, "ALT complex_expr = " // &
    "cexpr_real | cexpr_complex")
call ifile_append (ifile, "ARG cexpr_complex = ( expr, expr )")
call ifile_append (ifile, "SEQ cexpr_real = expr")
call ifile_append (ifile, "IDE var_name")
call ifile_append (ifile, "KEY '='")

```

```

call ifile_append (ifile, "KEY in")
call ifile_append (ifile, "SEQ conditional_expr = " // &
    "if lexpr then expr maybe_elseif_expr maybe_else_expr endif")
call ifile_append (ifile, "SEQ maybe_elseif_expr = elseif_expr*")
call ifile_append (ifile, "SEQ maybe_else_expr = else_expr*")
call ifile_append (ifile, "SEQ elseif_expr = elseif lexpr then expr")
call ifile_append (ifile, "SEQ else_expr = else expr")
call ifile_append (ifile, "KEY if")
call ifile_append (ifile, "KEY then")
call ifile_append (ifile, "KEY elseif")
call ifile_append (ifile, "KEY else")
call ifile_append (ifile, "KEY endif")
call define_lexpr_syntax (ifile, particles, analysis)
call define_sexpr_syntax (ifile)
if (particles) then
    call define_pexpr_syntax (ifile)
    call define_cexpr_syntax (ifile)
    call define_var_plist_syntax (ifile)
    call define_var_alias_syntax (ifile)
    call define_numeric_pexpr_syntax (ifile)
    call define_logical_pexpr_syntax (ifile)
end if

end subroutine define_expr_syntax

```

Logical expressions.

*(Expressions: procedures)*+≡

```

subroutine define_lexpr_syntax (ifile, particles, analysis)
    type(ifile_t), intent(inout) :: ifile
    logical, intent(in) :: particles, analysis
    type(string_t) :: logical_pexpr, record_cmd
    if (particles) then
        logical_pexpr = " | logical_pexpr"
    else
        logical_pexpr = ""
    end if
    if (analysis) then
        record_cmd = " | record_cmd"
    else
        record_cmd = ""
    end if
    call ifile_append (ifile, "SEQ lexpr = lsinglet lsequel*")
    call ifile_append (ifile, "SEQ lsequel = ';' lsinglet")
    call ifile_append (ifile, "SEQ lsinglet = lterm alternative*")
    call ifile_append (ifile, "SEQ alternative = or lterm")
    call ifile_append (ifile, "SEQ lterm = lvalue coincidence*")
    call ifile_append (ifile, "SEQ coincidence = and lvalue")
    call ifile_append (ifile, "KEY ';'")
    call ifile_append (ifile, "KEY or")
    call ifile_append (ifile, "KEY and")
    call ifile_append (ifile, "ALT lvalue = " // &
        "true | false | lvariable | negation | " // &
        "grouped_lexpr | block_lexpr | conditional_lexpr | " // &
        "compared_expr | compared_sexpr" // &

```

```

        logical_pexpr // record_cmd)
call ifile_append (ifile, "KEY true")
call ifile_append (ifile, "KEY false")
call ifile_append (ifile, "SEQ lvariable = '?' alt_lvariable")
call ifile_append (ifile, "KEY '?'")
call ifile_append (ifile, "ALT alt_lvariable = variable | grouped_lexpr")
call ifile_append (ifile, "SEQ negation = not lvalue")
call ifile_append (ifile, "KEY not")
call ifile_append (ifile, "GRO grouped_lexpr = ( lexpr )")
call ifile_append (ifile, "SEQ block_lexpr = let var_spec in lexpr")
call ifile_append (ifile, "ALT var_logical = " // &
    "var_logical_new | var_logical_spec")
call ifile_append (ifile, "SEQ var_logical_new = logical var_logical_spec")
call ifile_append (ifile, "KEY logical")
call ifile_append (ifile, "SEQ var_logical_spec = '?' var_name = lexpr")
call ifile_append (ifile, "SEQ conditional_lexpr = " // &
    "if lexpr then lexpr maybe_elseif_lexpr maybe_else_lexpr endif")
call ifile_append (ifile, "SEQ maybe_elseif_lexpr = elseif_lexpr*")
call ifile_append (ifile, "SEQ maybe_else_lexpr = else_lexpr?")
call ifile_append (ifile, "SEQ elseif_lexpr = elseif lexpr then lexpr")
call ifile_append (ifile, "SEQ else_lexpr = else lexpr")
call ifile_append (ifile, "SEQ compared_expr = expr comparison+")
call ifile_append (ifile, "SEQ comparison = compare expr")
call ifile_append (ifile, "ALT compare = " // &
    "<' | '> | '<=' | '>=' | '==' | '<>' | '==~' | '<>~'")
call ifile_append (ifile, "KEY '<'")
call ifile_append (ifile, "KEY '>'")
call ifile_append (ifile, "KEY '<='")
call ifile_append (ifile, "KEY '>='")
call ifile_append (ifile, "KEY '=='")
call ifile_append (ifile, "KEY '<>'")
call ifile_append (ifile, "KEY '==~'")
call ifile_append (ifile, "KEY '<>~'")
call ifile_append (ifile, "SEQ compared_sexpr = sexpr str_comparison+")
call ifile_append (ifile, "SEQ str_comparison = str_compare sexpr")
call ifile_append (ifile, "ALT str_compare = '==' | '<>'")
if (analysis) then
    call ifile_append (ifile, "SEQ record_cmd = " // &
        "record_key analysis_tag record_arg?")
    call ifile_append (ifile, "ALT record_key = record | record_unweighted")
    call ifile_append (ifile, "KEY record")
    call ifile_append (ifile, "KEY record_unweighted")
    call ifile_append (ifile, "ALT analysis_tag = analysis_id | sexpr")
    call ifile_append (ifile, "IDE analysis_id")
    call ifile_append (ifile, "ARG record_arg = ( expr+ )")
end if
end subroutine define_lexpr_syntax

```

String expressions.

*(Expressions: procedures)+≡*

```

subroutine define_sexpr_syntax (ifile)
type(ifile_t), intent(inout) :: ifile
call ifile_append (ifile, "SEQ sexpr = svalue str_concatenation*")
call ifile_append (ifile, "SEQ str_concatenation = '&' svalue")

```

```

call ifile_append (ifile, "KEY '&'" )
call ifile_append (ifile, "ALT svalue = " // &
    "grouped_sexpr | block_sexpr | conditional_sexpr | " // &
    "svariable | string_function | string_literal")
call ifile_append (ifile, "GRO grouped_sexpr = ( sexpr )")
call ifile_append (ifile, "SEQ block_sexpr = let var_spec in sexpr")
call ifile_append (ifile, "SEQ conditional_sexpr = " // &
    "if lexpr then sexpr maybe_elseif_sexpr maybe_else_sexpr endif")
call ifile_append (ifile, "SEQ maybe_elseif_sexpr = elseif_sexpr*")
call ifile_append (ifile, "SEQ maybe_else_sexpr = else_sexpr?")
call ifile_append (ifile, "SEQ elseif_sexpr = elseif lexpr then sexpr")
call ifile_append (ifile, "SEQ else_sexpr = else sexpr")
call ifile_append (ifile, "SEQ svariable = '$' alt_svariable")
call ifile_append (ifile, "KEY '$'")
call ifile_append (ifile, "ALT alt_svariable = variable | grouped_sexpr")
call ifile_append (ifile, "ALT var_string = " // &
    "var_string_new | var_string_spec")
call ifile_append (ifile, "SEQ var_string_new = string var_string_spec")
call ifile_append (ifile, "KEY string")
call ifile_append (ifile, "SEQ var_string_spec = '$' var_name = sexpr") ! $
call ifile_append (ifile, "ALT string_function = sprintf_fun | sprintf_fun")
call ifile_append (ifile, "SEQ sprintf_fun = sprintf sprintf_args?")
call ifile_append (ifile, "KEY sprintf")
call ifile_append (ifile, "SEQ sprintf_fun = sprintf_clause sprintf_args?")
call ifile_append (ifile, "SEQ sprintf_clause = sprintf sexpr")
call ifile_append (ifile, "KEY sprintf")
call ifile_append (ifile, "ARG sprintf_args = ( sprintf_arg* )")
call ifile_append (ifile, "ALT sprintf_arg = " &
    // "lvariable | svariable | expr")
call ifile_append (ifile, "QUO string_literal = '""'...'""'")
end subroutine define_sexpr_syntax

```

Expressions that evaluate to subevents.

*(Expressions: procedures)*+≡

```

subroutine define_pexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ pexpr = pterm pconcatenation*")
    call ifile_append (ifile, "SEQ pconcatenation = '&' pterm")
!   call ifile_append (ifile, "KEY '&'" )
    call ifile_append (ifile, "SEQ pterm = pvalue pcombination*")
    call ifile_append (ifile, "SEQ pcombination = '+' pvalue")
!   call ifile_append (ifile, "KEY '+'")
    call ifile_append (ifile, "ALT pvalue = " // &
        "pexpr_src | pvariable | " // &
        "grouped_pexpr | block_pexpr | conditional_pexpr | " // &
        "prt_function")
    call ifile_append (ifile, "SEQ pexpr_src = prefix_cexpr")
    call ifile_append (ifile, "ALT prefix_cexpr = " // &
        "incoming_prt | outgoing_prt")
    call ifile_append (ifile, "SEQ incoming_prt = incoming cexpr")
    call ifile_append (ifile, "KEY incoming")
    call ifile_append (ifile, "SEQ outgoing_prt = cexpr")
    call ifile_append (ifile, "SEQ pvariable = '@' alt_pvariable")
    call ifile_append (ifile, "KEY '@'")

```

```

call ifile_append (ifile, "ALT alt_pvariable = variable | grouped_pexpr")
call ifile_append (ifile, "GRO grouped_pexpr = '[' pexpr ']'")
call ifile_append (ifile, "SEQ block_pexpr = let var_spec in pexpr")
call ifile_append (ifile, "SEQ conditional_pexpr = " // &
    "if lexpr then pexpr maybe_elseif_pexpr maybe_else_pexpr endif")
call ifile_append (ifile, "SEQ maybe_elseif_pexpr = elseif_pexpr*")
call ifile_append (ifile, "SEQ maybe_else_pexpr = else_pexpr?")
call ifile_append (ifile, "SEQ elseif_pexpr = elseif lexpr then pexpr")
call ifile_append (ifile, "SEQ else_pexpr = else pexpr")
call ifile_append (ifile, "ALT prt_function = " // &
    "join_fun | combine_fun | collect_fun | select_fun | " // &
    "extract_fun | sort_fun")
call ifile_append (ifile, "SEQ join_fun = join_clause pargs2")
call ifile_append (ifile, "SEQ combine_fun = combine_clause pargs2")
call ifile_append (ifile, "SEQ collect_fun = collect_clause pargs1")
call ifile_append (ifile, "SEQ select_fun = select_clause pargs1")
call ifile_append (ifile, "SEQ extract_fun = extract_clause pargs1")
call ifile_append (ifile, "SEQ sort_fun = sort_clause pargs1")
call ifile_append (ifile, "SEQ join_clause = join condition?")
call ifile_append (ifile, "SEQ combine_clause = combine condition?")
call ifile_append (ifile, "SEQ collect_clause = collect condition?")
call ifile_append (ifile, "SEQ select_clause = select condition?")
call ifile_append (ifile, "SEQ extract_clause = extract position?")
call ifile_append (ifile, "SEQ sort_clause = sort criterion?")
call ifile_append (ifile, "KEY join")
call ifile_append (ifile, "KEY combine")
call ifile_append (ifile, "KEY collect")
call ifile_append (ifile, "KEY select")
call ifile_append (ifile, "SEQ condition = if lexpr")
call ifile_append (ifile, "KEY extract")
call ifile_append (ifile, "SEQ position = index expr")
call ifile_append (ifile, "KEY sort")
call ifile_append (ifile, "SEQ criterion = by expr")
call ifile_append (ifile, "KEY index")
call ifile_append (ifile, "KEY by")
call ifile_append (ifile, "ARG pargs2 = '[' pexpr, pexpr ']'")
call ifile_append (ifile, "ARG pargs1 = '[' pexpr, pexpr? ']'")
end subroutine define_pexpr_syntax

```

Expressions that evaluate to PDG-code arrays.

*(Expressions: procedures)* +≡

```

subroutine define_cexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ cexpr = avalue concatenation*")
    call ifile_append (ifile, "SEQ concatenation = ':' avalue")
    call ifile_append (ifile, "KEY ':'")
    call ifile_append (ifile, "ALT avalue = " // &
        "grouped_cexpr | block_cexpr | conditional_cexpr | " // &
        "variable | pdg_code | prt_name")
    call ifile_append (ifile, "GRO grouped_cexpr = ( cexpr )")
    call ifile_append (ifile, "SEQ block_cexpr = let var_spec in cexpr")
    call ifile_append (ifile, "SEQ conditional_cexpr = " // &
        "if lexpr then cexpr maybe_elseif_cexpr maybe_else_cexpr endif")
    call ifile_append (ifile, "SEQ maybe_elseif_cexpr = elseif_cexpr*")

```



```

call ifile_append (ifile, "SEQ maybe_else_cexpr = else_cexpr?")
call ifile_append (ifile, "SEQ elsif_cexpr = elsif lexpr then cexpr")
call ifile_append (ifile, "SEQ else_cexpr = else cexpr")
call ifile_append (ifile, "SEQ pdg_code = pdg pdg_arg")
call ifile_append (ifile, "KEY pdg")
call ifile_append (ifile, "ARG pdg_arg = ( expr )")
call ifile_append (ifile, "QUO prt_name = '""'...'""')
end subroutine define_cexpr_syntax

```

Extra variable types.

*(Expressions: procedures)*+≡

```

subroutine define_var_plist_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "ALT var_plist = var_plist_new | var_plist_spec")
  call ifile_append (ifile, "SEQ var_plist_new = subevt var_plist_spec")
  call ifile_append (ifile, "KEY subevt")
  call ifile_append (ifile, "SEQ var_plist_spec = '@' var_name '=' pexpr")
end subroutine define_var_plist_syntax

subroutine define_var_alias_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ var_alias = alias var_name '=' cexpr")
  call ifile_append (ifile, "KEY alias")
end subroutine define_var_alias_syntax

```

Particle-list expressions that evaluate to numeric values

*(Expressions: procedures)*+≡

```

subroutine define_numeric_pexpr_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "ALT numeric_pexpr = " &
    // "eval_fun | count_fun | event_shape_fun")
  call ifile_append (ifile, "SEQ eval_fun = eval expr pargs1")
  call ifile_append (ifile, "SEQ count_fun = count_clause pargs1")
  call ifile_append (ifile, "SEQ count_clause = count condition?")
  call ifile_append (ifile, "KEY eval")
  call ifile_append (ifile, "KEY count")
  call ifile_append (ifile, "ALT event_shape_fun = user_event_fun")
  call ifile_append (ifile, "SEQ user_event_fun = " &
    // "user_event_shape user_arg pargs1")
  call ifile_append (ifile, "KEY user_event_shape")
end subroutine define_numeric_pexpr_syntax

```

Particle-list functions that evaluate to logical values.

*(Expressions: procedures)*+≡

```

subroutine define_logical_pexpr_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "ALT logical_pexpr = " // &
    "all_fun | any_fun | no_fun | user_cut_fun")
  call ifile_append (ifile, "SEQ all_fun = all lexpr pargs1")
  call ifile_append (ifile, "SEQ any_fun = any lexpr pargs1")
  call ifile_append (ifile, "SEQ no_fun = no lexpr pargs1")
  call ifile_append (ifile, "KEY all")

```

```

call ifile_append (ifile, "KEY any")
call ifile_append (ifile, "KEY no")
call ifile_append (ifile, "SEQ user_cut_fun = user_cut user_arg pargs1")
call ifile_append (ifile, "KEY user_cut")
end subroutine define_logical_pexpr_syntax

```

All characters that can occur in expressions (apart from alphanumeric).

```

<Expressions: procedures>+≡
subroutine lexer_init_eval_tree (lexer, particles)
  type(lexer_t), intent(out) :: lexer
  logical, intent(in) :: particles
  type(keyword_list_t), pointer :: keyword_list
  if (particles) then
    keyword_list => syntax_get_keyword_list_ptr (syntax_pexpr)
  else
    keyword_list => syntax_get_keyword_list_ptr (syntax_expr)
  end if
  call lexer_init (lexer, &
    comment_chars = "#!", &
    quote_chars = "'", &
    quote_match = "'", &
    single_chars = "()[];:&%?${}@", &
    special_class = (/ "+-*/^", "<>=~ " /) , &
    keyword_list = keyword_list)
end subroutine lexer_init_eval_tree

```

### 5.6.8 Set up appropriate parse trees

Parse an input stream as a specific flavor of expression. The appropriate expression syntax has to be available.

```

<Expressions: public>+≡
public :: parse_tree_init_expr
public :: parse_tree_init_lexpr
public :: parse_tree_init_pexpr
public :: parse_tree_init_cexpr
public :: parse_tree_init_sexpr

<Expressions: procedures>+≡
subroutine parse_tree_init_expr (parse_tree, stream, particles)
  type(parse_tree_t), intent(out) :: parse_tree
  type(stream_t), intent(inout), target :: stream
  logical, intent(in) :: particles
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, particles)
  call lexer_assign_stream (lexer, stream)
  if (particles) then
    call parse_tree_init &
      (parse_tree, syntax_pexpr, lexer, var_str ("expr"))
  else
    call parse_tree_init &
      (parse_tree, syntax_expr, lexer, var_str ("expr"))
  end if
end subroutine parse_tree_init_expr

```

```

        call lexer_final (lexer)
    end subroutine parse_tree_init_expr

subroutine parse_tree_init_lexpr (parse_tree, stream, particles)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    logical, intent(in) :: particles
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, particles)
    call lexer_assign_stream (lexer, stream)
    if (particles) then
        call parse_tree_init &
            (parse_tree, syntax_pexpr, lexer, var_str ("lexpr"))
    else
        call parse_tree_init &
            (parse_tree, syntax_expr, lexer, var_str ("lexpr"))
    end if
    call lexer_final (lexer)
end subroutine parse_tree_init_lexpr

subroutine parse_tree_init_pexpr (parse_tree, stream)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, .true.)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init &
        (parse_tree, syntax_pexpr, lexer, var_str ("pexpr"))
    call lexer_final (lexer)
end subroutine parse_tree_init_pexpr

subroutine parse_tree_init_cexpr (parse_tree, stream)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, .true.)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init &
        (parse_tree, syntax_pexpr, lexer, var_str ("cexpr"))
    call lexer_final (lexer)
end subroutine parse_tree_init_cexpr

subroutine parse_tree_init_sexpr (parse_tree, stream, particles)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    logical, intent(in) :: particles
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, particles)
    call lexer_assign_stream (lexer, stream)
    if (particles) then
        call parse_tree_init &
            (parse_tree, syntax_pexpr, lexer, var_str ("sexpr"))
    else
        call parse_tree_init &

```

```

        (parse_tree, syntax_expr, lexer, var_str ("sexpr"))
    end if
    call lexer_final (lexer)
end subroutine parse_tree_init_sexpr

```

### 5.6.9 The evaluation tree

The evaluation tree contains the initial variable list and the root node.

```

<Expressions: public>+≡
    public :: eval_tree_t

<Expressions: types>+≡
    type :: eval_tree_t
        private
        type(var_list_t) :: var_list
        type(eval_node_t), pointer :: root => null ()
    contains
        <Expressions: eval tree: TBP>
    end type eval_tree_t

```

Init from stream, using a temporary parse tree.

```

<Expressions: procedures>+≡
    subroutine eval_tree_init_stream &
        (eval_tree, stream, var_list, subevt, result_type)
        type(eval_tree_t), intent(out), target :: eval_tree
        type(stream_t), intent(inout), target :: stream
        type(var_list_t), intent(in), target :: var_list
        type(subevt_t), intent(in), target, optional :: subevt
        integer, intent(in), optional :: result_type
        type(parse_tree_t) :: parse_tree
        type(parse_node_t), pointer :: nd_root
        integer :: type
        type = V_REAL; if (present (result_type)) type = result_type
        select case (type)
        case (V_INT, V_REAL, V_CMPLX)
            call parse_tree_init_expr (parse_tree, stream, present (subevt))
        case (V_LOG)
            call parse_tree_init_lexpr (parse_tree, stream, present (subevt))
        case (V_SEV)
            call parse_tree_init_pexpr (parse_tree, stream)
        case (V_PDG)
            call parse_tree_init_cexpr (parse_tree, stream)
        case (V_STR)
            call parse_tree_init_sexpr (parse_tree, stream, present (subevt))
        end select
        call parse_tree_write (parse_tree)
        nd_root => parse_tree_get_root_ptr (parse_tree)
        if (associated (nd_root)) then
            select case (type)
            case (V_INT, V_REAL, V_CMPLX)
                call eval_tree_init_expr (eval_tree, nd_root, var_list, subevt)
            case (V_LOG)

```

```

        call eval_tree_init_lexpr (eval_tree, nd_root, var_list, subevt)
    case (V_SEV)
        call eval_tree_init_pexpr (eval_tree, nd_root, var_list, subevt)
    case (V_PDG)
        call eval_tree_init_cexpr (eval_tree, nd_root, var_list, subevt)
    case (V_STR)
        call eval_tree_init_sexpr (eval_tree, nd_root, var_list, subevt)
    end select
end if
call parse_tree_final (parse_tree)
end subroutine eval_tree_init_stream

```

API: Init from a given parse-tree node. If we evaluate an expression that contains particle-list references, the original subevent has to be supplied. The initial variable list is optional.

*(Expressions: public)*+≡

```

public :: eval_tree_init_expr
public :: eval_tree_init_lexpr
public :: eval_tree_init_pexpr
public :: eval_tree_init_cexpr
public :: eval_tree_init_sexpr

```

*(Expressions: procedures)*+≡

```

subroutine eval_tree_init_expr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    call eval_tree_set_var_list (eval_tree, var_list, subevt, event_vars)
    call eval_node_compile_expr &
        (eval_tree%root, parse_node, eval_tree%var_list)
end subroutine eval_tree_init_expr

subroutine eval_tree_init_lexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    call eval_tree_set_var_list (eval_tree, var_list, subevt, event_vars)
    call eval_node_compile_lexpr &
        (eval_tree%root, parse_node, eval_tree%var_list)
end subroutine eval_tree_init_lexpr

subroutine eval_tree_init_pexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars

```

```

    call eval_tree_set_var_list (eval_tree, var_list, subevt, event_vars)
    call eval_node_compile_pexpr &
        (eval_tree%root, parse_node, eval_tree%var_list)
end subroutine eval_tree_init_pexpr

subroutine eval_tree_init_cexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    call eval_tree_set_var_list (eval_tree, var_list, subevt, event_vars)
    call eval_node_compile_cexpr &
        (eval_tree%root, parse_node, eval_tree%var_list)
end subroutine eval_tree_init_cexpr

subroutine eval_tree_init_sexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    call eval_tree_set_var_list (eval_tree, var_list, subevt, event_vars)
    call eval_node_compile_sexpr &
        (eval_tree%root, parse_node, eval_tree%var_list)
end subroutine eval_tree_init_sexpr

```

This extra API function handles numerical constant expressions only. The only nontrivial part is the optional unit.

```

<Expressions: public>+≡
    public :: eval_tree_init_numeric_value

<Expressions: procedures>+≡
    subroutine eval_tree_init_numeric_value (eval_tree, parse_node)
        type(eval_tree_t), intent(out), target :: eval_tree
        type(parse_node_t), intent(in), target :: parse_node
        call eval_node_compile_numeric_value (eval_tree%root, parse_node)
    end subroutine eval_tree_init_numeric_value

```

Initialize the variable list with the initial one; if a particle list is provided, add a pointer to this as variable @evt. If the event weight is provided as a real-valued target, add a pointer to it as well.

```

<Expressions: procedures>+≡
    subroutine eval_tree_set_var_list &
        (eval_tree, var_list, subevt, event_vars)
        type(eval_tree_t), intent(inout), target :: eval_tree
        type(var_list_t), intent(in), target :: var_list
        type(subevt_t), intent(in), optional, target :: subevt
        type(event_vars_t), intent(in), optional, target :: event_vars
        logical, save, target :: known = .true.
        call var_list_link (eval_tree%var_list, var_list)
    end subroutine eval_tree_set_var_list

```

```

    if (present (subevt)) call var_list_append_subevt_ptr &
        (eval_tree%var_list, var_str ("@evt"), subevt, known, &
        intrinsic=.true.)
    if (present (event_vars)) &
        call var_list_append_event_vars (eval_tree%var_list, event_vars)
end subroutine eval_tree_set_var_list

```

*<Expressions: public>+≡*  
 public :: eval\_tree\_final

*<Expressions: procedures>+≡*  
 subroutine eval\_tree\_final (eval\_tree)  
 type(eval\_tree\_t), intent(inout) :: eval\_tree  
 call var\_list\_final (eval\_tree%var\_list)  
 if (associated (eval\_tree%root)) then  
 call eval\_node\_final\_rec (eval\_tree%root)  
 deallocate (eval\_tree%root)  
 end if  
 end subroutine eval\_tree\_final

*<Expressions: public>+≡*  
 public :: eval\_tree\_evaluate

*<Expressions: procedures>+≡*  
 subroutine eval\_tree\_evaluate (eval\_tree)  
 type(eval\_tree\_t), intent(inout) :: eval\_tree  
 if (associated (eval\_tree%root)) then  
 call eval\_node\_evaluate (eval\_tree%root)  
 end if  
 end subroutine eval\_tree\_evaluate

Check if the eval tree is allocated.

*<Expressions: public>+≡*  
 public :: eval\_tree\_is\_defined

*<Expressions: procedures>+≡*  
 function eval\_tree\_is\_defined (eval\_tree) result (flag)  
 logical :: flag  
 type(eval\_tree\_t), intent(in) :: eval\_tree  
 flag = associated (eval\_tree%root)  
 end function eval\_tree\_is\_defined

Check if the eval tree result is constant.

*<Expressions: public>+≡*  
 public :: eval\_tree\_is\_constant

*<Expressions: procedures>+≡*  
 function eval\_tree\_is\_constant (eval\_tree) result (flag)  
 logical :: flag  
 type(eval\_tree\_t), intent(in) :: eval\_tree  
 if (associated (eval\_tree%root)) then  
 flag = eval\_tree%root%type == EN\_CONSTANT  
 else  
 flag = .false.  
 end if  
 end function eval\_tree\_is\_constant

```

        end if
    end function eval_tree_is_constant

```

Insert a conversion node at the root, if necessary (only for real/int conversion)

```

<Expressions: public>+≡
    public :: eval_tree_convert_result

<Expressions: procedures>+≡
    subroutine eval_tree_convert_result (eval_tree, result_type)
        type(eval_tree_t), intent(inout) :: eval_tree
        integer, intent(in) :: result_type
        if (associated (eval_tree%root)) then
            call insert_conversion_node (eval_tree%root, result_type)
        end if
    end subroutine eval_tree_convert_result

```

Return the value of the top node, after evaluation. If the tree is empty, return the type of V\_NONE. When extracting the value, no check for existence is done. For numeric values, the functions are safe against real/integer mismatch.

```

<Expressions: public>+≡
    public :: eval_tree_get_result_type
    public :: eval_tree_result_is_known
    public :: eval_tree_result_is_known_ptr
    public :: eval_tree_get_log
    public :: eval_tree_get_int
    public :: eval_tree_get_real
    public :: eval_tree_get_cmplx
    public :: eval_tree_get_pdg_array
    public :: eval_tree_get_subevt
    public :: eval_tree_get_string

<Expressions: procedures>+≡
    function eval_tree_get_result_type (eval_tree) result (type)
        integer :: type
        type(eval_tree_t), intent(in) :: eval_tree
        if (associated (eval_tree%root)) then
            type = eval_tree%root%result_type
        else
            type = V_NONE
        end if
    end function eval_tree_get_result_type

    function eval_tree_result_is_known (eval_tree) result (flag)
        logical :: flag
        type(eval_tree_t), intent(in) :: eval_tree
        if (associated (eval_tree%root)) then
            select case (eval_tree%root%result_type)
            case (V_LOG, V_INT, V_REAL)
                flag = eval_tree%root%value_is_known
            case default
                flag = .true.
            end select
        else
            flag = .false.
        end if
    end function eval_tree_result_is_known

```



```

end if
end function eval_tree_result_is_known

function eval_tree_result_is_known_ptr (eval_tree) result (ptr)
    logical, pointer :: ptr
    type(eval_tree_t), intent(in) :: eval_tree
    logical, target, save :: known = .true.
    if (associated (eval_tree%root)) then
        select case (eval_tree%root%result_type)
            case (V_LOG, V_INT, V_REAL)
                ptr => eval_tree%root%value_is_known
            case default
                ptr => known
        end select
    else
        ptr => null ()
    end if
end function eval_tree_result_is_known_ptr

function eval_tree_get_log (eval_tree) result (lval)
    logical :: lval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) lval = eval_tree%root%lval
end function eval_tree_get_log

function eval_tree_get_int (eval_tree) result (ival)
    integer :: ival
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        select case (eval_tree%root%result_type)
            case (V_INT); ival = eval_tree%root%ival
            case (V_REAL); ival = eval_tree%root%rval
            case (V_CMPLX); ival = eval_tree%root%cval
        end select
    end if
end function eval_tree_get_int

function eval_tree_get_real (eval_tree) result (rval)
    real(default) :: rval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        select case (eval_tree%root%result_type)
            case (V_REAL); rval = eval_tree%root%rval
            case (V_INT); rval = eval_tree%root%ival
            case (V_CMPLX); rval = eval_tree%root%cval
        end select
    end if
end function eval_tree_get_real

function eval_tree_get_cmplx (eval_tree) result (cval)
    complex(default) :: cval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        select case (eval_tree%root%result_type)

```

```

        case (V_CMPLX); cval = eval_tree%root%cval
        case (V_REAL); cval = eval_tree%root%rval
        case (V_INT); cval = eval_tree%root%ival
    end select
end if
end function eval_tree_get_cmplx

function eval_tree_get_pdg_array (eval_tree) result (aval)
    type(pdg_array_t) :: aval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        aval = eval_tree%root%aval
    end if
end function eval_tree_get_pdg_array

function eval_tree_get_subevt (eval_tree) result (pval)
    type(subevt_t) :: pval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        pval = eval_tree%root%pval
    end if
end function eval_tree_get_subevt

function eval_tree_get_string (eval_tree) result (sval)
    type(string_t) :: sval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        sval = eval_tree%root%sval
    end if
end function eval_tree_get_string

```

Return a pointer to the value of the top node.

```

<Expressions: public>+≡
    public :: eval_tree_get_log_ptr
    public :: eval_tree_get_int_ptr
    public :: eval_tree_get_real_ptr
    public :: eval_tree_get_cmplx_ptr
    public :: eval_tree_get_subevt_ptr
    public :: eval_tree_get_pdg_array_ptr
    public :: eval_tree_get_string_ptr

<Expressions: procedures>+≡
    function eval_tree_get_log_ptr (eval_tree) result (lval)
        logical, pointer :: lval
        type(eval_tree_t), intent(in) :: eval_tree
        if (associated (eval_tree%root)) then
            lval => eval_tree%root%lval
        else
            lval => null ()
        end if
    end function eval_tree_get_log_ptr

    function eval_tree_get_int_ptr (eval_tree) result (ival)
        integer, pointer :: ival

```

```

    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        ival => eval_tree%root%ival
    else
        ival => null ()
    end if
end function eval_tree_get_int_ptr

function eval_tree_get_real_ptr (eval_tree) result (rval)
    real(default), pointer :: rval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        rval => eval_tree%root%rval
    else
        rval => null ()
    end if
end function eval_tree_get_real_ptr

function eval_tree_get_cmplx_ptr (eval_tree) result (cval)
    complex(default), pointer :: cval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        cval => eval_tree%root%cval
    else
        cval => null ()
    end if
end function eval_tree_get_cmplx_ptr

function eval_tree_get_subevt_ptr (eval_tree) result (pval)
    type(subevt_t), pointer :: pval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        pval => eval_tree%root%pval
    else
        pval => null ()
    end if
end function eval_tree_get_subevt_ptr

function eval_tree_get_pdg_array_ptr (eval_tree) result (aval)
    type(pdg_array_t), pointer :: aval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        aval => eval_tree%root%aval
    else
        aval => null ()
    end if
end function eval_tree_get_pdg_array_ptr

function eval_tree_get_string_ptr (eval_tree) result (sval)
    type(string_t), pointer :: sval
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
        sval => eval_tree%root%sval
    else

```

```

        sval => null ()
    end if
end function eval_tree_get_string_ptr

<Expressions: public>+≡
    public :: eval_tree_write

<Expressions: eval tree: TBP>≡
    procedure :: write => eval_tree_write

<Expressions: procedures>+≡
    subroutine eval_tree_write (eval_tree, unit, write_var_list)
        class(eval_tree_t), intent(in) :: eval_tree
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: write_var_list
        integer :: u
        logical :: vl
        u = output_unit (unit); if (u < 0) return
        vl = .false.; if (present (write_var_list)) vl = write_var_list
        write (u, "(1x,A)") "Evaluation tree:"
        if (associated (eval_tree%root)) then
            call eval_node_write_rec (eval_tree%root, unit)
        else
            write (u, "(3x,A)") "[empty]"
        end if
        if (vl) call var_list_write (eval_tree%var_list, unit)
    end subroutine eval_tree_write

```

Use the written representation for generating an MD5 sum:

```

<Expressions: public>+≡
    public :: eval_tree_get_md5sum

<Expressions: procedures>+≡
    function eval_tree_get_md5sum (eval_tree) result (md5sum_et)
        character(32) :: md5sum_et
        type(eval_tree_t), intent(in) :: eval_tree
        integer :: u
        u = free_unit ()
        open (unit = u, status = "scratch", action = "readwrite")
        call eval_tree_write (eval_tree, unit=u)
        rewind (u)
        md5sum_et = md5sum (u)
        close (u)
    end function eval_tree_get_md5sum

```

### 5.6.10 Direct evaluation

These procedures create an eval tree and evaluate it on-the-fly, returning only the final value. The evaluation must yield a well-defined value, unless the `is_known` flag is present, which will be set accordingly.

```

<Expressions: public>+≡
    public :: eval_log
    public :: eval_int

```

```

public :: eval_real
public :: eval_cmplx
public :: eval_subevt
public :: eval_pdg_array
public :: eval_string

<Expressions: procedures>+≡
function eval_log &
    (parse_node, var_list, subevt, event_vars, is_known) result (lval)
    logical :: lval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_lexpr &
        (eval_tree, parse_node, var_list, subevt, event_vars)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
        if (present (is_known)) is_known = .true.
        lval = eval_tree_get_log (eval_tree)
    else if (present (is_known)) then
        is_known = .false.
    else
        call eval_tree_unknown (eval_tree, parse_node)
        lval = .false.
    end if
    call eval_tree_final (eval_tree)
end function eval_log

function eval_int &
    (parse_node, var_list, subevt, event_vars, is_known) result (ival)
    integer :: ival
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_expr &
        (eval_tree, parse_node, var_list, subevt, event_vars)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
        if (present (is_known)) is_known = .true.
        ival = eval_tree_get_int (eval_tree)
    else if (present (is_known)) then
        is_known = .false.
    else
        call eval_tree_unknown (eval_tree, parse_node)
        ival = 0
    end if
    call eval_tree_final (eval_tree)
end function eval_int

```

```

function eval_real &
    (parse_node, var_list, subevt, event_vars, is_known) result (rval)
    real(default) :: rval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_expr &
        (eval_tree, parse_node, var_list, subevt, event_vars)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
        if (present (is_known)) is_known = .true.
        rval = eval_tree_get_real (eval_tree)
    else if (present (is_known)) then
        is_known = .false.
    else
        call eval_tree_unknown (eval_tree, parse_node)
        rval = 0
    end if
    call eval_tree_final (eval_tree)
end function eval_real

function eval_cmplx &
    (parse_node, var_list, subevt, event_vars, is_known) result (cval)
    complex(default) :: cval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    type(event_vars_t), intent(in), optional, target :: event_vars
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_expr &
        (eval_tree, parse_node, var_list, subevt, event_vars)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
        if (present (is_known)) is_known = .true.
        cval = eval_tree_get_cmplx (eval_tree)
    else if (present (is_known)) then
        is_known = .false.
    else
        call eval_tree_unknown (eval_tree, parse_node)
        cval = 0
    end if
    call eval_tree_final (eval_tree)
end function eval_cmplx

function eval_subevt &
    (parse_node, var_list, subevt, event_vars, is_known) result (pval)
    type(subevt_t) :: pval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt

```

```

type(event_vars_t), intent(in), optional, target :: event_vars
logical, intent(out), optional :: is_known
type(eval_tree_t), target :: eval_tree
call eval_tree_init_pexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
call eval_tree_evaluate (eval_tree)
if (eval_tree_result_is_known (eval_tree)) then
    if (present (is_known)) is_known = .true.
    pval = eval_tree_get_subevt (eval_tree)
else if (present (is_known)) then
    is_known = .false.
else
    call eval_tree_unknown (eval_tree, parse_node)
end if
call eval_tree_final (eval_tree)
end function eval_subevt

function eval_pdg_array &
    (parse_node, var_list, subevt, event_vars, is_known) result (aval)
type(pdg_array_t) :: aval
type(parse_node_t), intent(in), target :: parse_node
type(var_list_t), intent(in), target :: var_list
type(subevt_t), intent(in), optional, target :: subevt
type(event_vars_t), intent(in), optional, target :: event_vars
logical, intent(out), optional :: is_known
type(eval_tree_t), target :: eval_tree
call eval_tree_init_cexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
call eval_tree_evaluate (eval_tree)
if (eval_tree_result_is_known (eval_tree)) then
    if (present (is_known)) is_known = .true.
    aval = eval_tree_get_pdg_array (eval_tree)
else if (present (is_known)) then
    is_known = .false.
else
    call eval_tree_unknown (eval_tree, parse_node)
end if
call eval_tree_final (eval_tree)
end function eval_pdg_array

function eval_string &
    (parse_node, var_list, subevt, event_vars, is_known) result (sval)
type(string_t) :: sval
type(parse_node_t), intent(in), target :: parse_node
type(var_list_t), intent(in), target :: var_list
type(subevt_t), intent(in), optional, target :: subevt
type(event_vars_t), intent(in), optional, target :: event_vars
logical, intent(out), optional :: is_known
type(eval_tree_t), target :: eval_tree
call eval_tree_init_sexpr &
    (eval_tree, parse_node, var_list, subevt, event_vars)
call eval_tree_evaluate (eval_tree)
if (eval_tree_result_is_known (eval_tree)) then
    if (present (is_known)) is_known = .true.

```

```

        sval = eval_tree_get_string (eval_tree)
    else if (present (is_known)) then
        is_known = .false.
    else
        call eval_tree_unknown (eval_tree, parse_node)
        sval = ""
    end if
    call eval_tree_final (eval_tree)
end function eval_string

```

Here is a variant that returns numeric values of all possible kinds, the appropriate kind to be selected later:

```

<Expressions: public>+≡
    public :: eval_numeric

<Expressions: procedures>+≡
    subroutine eval_numeric &
        (parse_node, var_list, subevt, event_vars, ival, rval, cval, &
         is_known, result_type)
        type(parse_node_t), intent(in), target :: parse_node
        type(var_list_t), intent(in), target :: var_list
        type(subevt_t), intent(in), optional, target :: subevt
        type(event_vars_t), intent(in), optional, target :: event_vars
        integer, intent(out), optional :: ival
        real(default), intent(out), optional :: rval
        complex(default), intent(out), optional :: cval
        logical, intent(out), optional :: is_known
        integer, intent(out), optional :: result_type
        type(eval_tree_t), target :: eval_tree
        call eval_tree_init_expr &
            (eval_tree, parse_node, var_list, subevt, event_vars)
        call eval_tree_evaluate (eval_tree)
        if (eval_tree_result_is_known (eval_tree)) then
            if (present (ival)) ival = eval_tree_get_int (eval_tree)
            if (present (rval)) rval = eval_tree_get_real (eval_tree)
            if (present (cval)) cval = eval_tree_get_cmplx (eval_tree)
            if (present (is_known)) is_known = .true.
        else
            call eval_tree_unknown (eval_tree, parse_node)
            if (present (ival)) ival = 0
            if (present (rval)) rval = 0
            if (present (cval)) cval = 0
            if (present (is_known)) is_known = .false.
        end if
        if (present (result_type)) &
            result_type = eval_tree_get_result_type (eval_tree)
        call eval_tree_final (eval_tree)
    end subroutine eval_numeric

```

Error message with debugging info:

```

<Expressions: procedures>+≡
    subroutine eval_tree_unknown (eval_tree, parse_node)
        type(eval_tree_t), intent(in) :: eval_tree

```



```

type(parse_node_t), intent(in) :: parse_node
call parse_node_write_rec (parse_node)
call eval_tree_write (eval_tree)
call msg_error ("Evaluation yields an undefined result, inserting default")
end subroutine eval_tree_unknown

```

### 5.6.11 Test

*<Expressions: public>+≡*

```
public :: expressions_test
```

*<Expressions: procedures>+≡*

```

subroutine expressions_test ()
  print *
  print *, "Expressions test 1"
  print *
  call expressions_test1 ()
  print *
  print *, "Expressions test 2"
  print *
  call syntax_expr_init ()
  call syntax_write (syntax_expr)
  call expressions_test2
  call syntax_expr_final ()
  print *
  print *, "Expressions test 3"
  print *
  call expressions_test3
  print *
  print *, "Expressions test 4"
  print *
  call syntax_pexpr_init ()
  call syntax_write (syntax_pexpr)
  call expressions_test4
  call syntax_pexpr_final ()
end subroutine expressions_test

subroutine expressions_test1 ()
  type(var_list_t), pointer :: var_list => null ()
  type(eval_node_t), pointer :: node => null ()
  type(prt_t), pointer :: prt => null ()
  type(var_entry_t), pointer :: var => null ()
  allocate (var_list)
  allocate (prt)
  call var_list_set_observables_unary (var_list, prt)
  call var_list_write (var_list)
  var => var_list_get_var_ptr (var_list, var_str ("PDG"))
  allocate (node)
  call eval_node_init_obs (node, var)
  call eval_node_write (node)
  call eval_node_final_rec (node)
  deallocate (node)
  call var_list_final (var_list)

```

```

deallocate (var_list)
deallocate (prt)
end subroutine expressions_test1

subroutine expressions_test2 ()
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(eval_tree_t) :: eval_tree
  type(string_t) :: expr_text
  type(var_list_t), pointer :: var_list => null ()
  allocate (var_list)
  call var_list_append_real (var_list, var_str ("x"), -5._default)
  call var_list_append_int (var_list, var_str ("foo"), -27)
  call var_list_append_real (var_list, var_str ("mb"), 4._default)
  expr_text = &
    "let real twopi = 2 * pi in" // &
    " twopi * sqrt (25.d0 - mb^2)" // &
    " / (let int mb_or_0 = max (mb, 0) in" // &
    "      1 + (if -1 TeV <= x < mb_or_0 then abs(x) else x endif))"
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call var_list_write (var_list)
  call eval_tree_init_stream (eval_tree, stream, var_list=var_list)
  call eval_tree_evaluate (eval_tree)
  call eval_tree_write (eval_tree)
  print "(A)", "Input string:"
  print *, char (expr_text)
  call stream_final (stream)
  call ifile_final (ifile)
  call eval_tree_final (eval_tree)
  call var_list_final (var_list)
  deallocate (var_list)
end subroutine expressions_test2

subroutine expressions_test3 ()
  type(subevt_t) :: subevt
  call subevt_init (subevt)
  call subevt_reset (subevt, 1)
  call subevt_set_incoming (subevt, 1, &
    22, vector4_moving (1.e3_default, 1.e3_default, 1), &
    0._default, (/ 2 /))
  call subevt_write (subevt)
  call subevt_reset (subevt, 4)
  call subevt_reset (subevt, 3)
  call subevt_set_incoming (subevt, 1, &
    21, vector4_moving (1.e3_default, 1.e3_default, 3), &
    0._default, (/ 1 /))
  call subevt_polarize (subevt, 1, -1)
  call subevt_set_outgoing (subevt, 2, &
    1, vector4_moving (0._default, 1.e3_default, 3), &
    -1.e6_default, (/ 7 /))
  call subevt_set_composite (subevt, 3, &
    vector4_moving (-1.e3_default, 0._default, 3), &
    (/ 2, 7 /))

```

```

    call subevt_write (subevt)
end subroutine expressions_test3

subroutine expressions_test4 ()
  type(subevt_t), target :: subevt
  type(string_t) :: expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(eval_tree_t) :: eval_tree
  type(var_list_t), pointer :: var_list => null ()
  type(pdg_array_t) :: aval
  allocate (var_list)
  aval = 0
  call var_list_append_pdg_array (var_list, var_str ("particle"), aval)
  aval = (/ 11,-11 /)
  call var_list_append_pdg_array (var_list, var_str ("lepton"), aval)
  aval = 22
  call var_list_append_pdg_array (var_list, var_str ("photon"), aval)
  aval = 1
  call var_list_append_pdg_array (var_list, var_str ("u"), aval)
  call subevt_init (subevt)
  call subevt_reset (subevt, 6)
  call subevt_set_incoming (subevt, 1, &
    1, vector4_moving (1._default, 1._default, 1), 0._default)
  call subevt_set_incoming (subevt, 2, &
    -1, vector4_moving (2._default, 2._default, 1), 0._default)
  call subevt_set_outgoing (subevt, 3, &
    22, vector4_moving (3._default, 3._default, 1), 0._default)
  call subevt_set_outgoing (subevt, 4, &
    22, vector4_moving (4._default, 4._default, 1), 0._default)
  call subevt_set_outgoing (subevt, 5, &
    11, vector4_moving (5._default, 5._default, 1), 0._default)
  call subevt_set_outgoing (subevt, 6, &
    -11, vector4_moving (6._default, 6._default, 1), 0._default)
  print *
  print *, "Expression:"
  expr_text = &
    "let alias quark = pdg(1):pdg(2):pdg(3) in" // &
    "  any E > 3 GeV " // &
    "    [sort by - Pt " // &
    "      [select if Index < 6 " // &
    "        [photon:pdg(-11):pdg(3):quark " // &
    "          & incoming particle]]]" // &
    " and" // &
    " eval Theta [extract index -1 [photon]] > 45 degree" // &
    " and" // &
    " count [incoming photon] * 3 > 0"
  print *, char (expr_text)
  print *
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call eval_tree_init_stream (eval_tree, stream, var_list, subevt, V_LOG)
  print *
  call eval_tree_write (eval_tree)

```

```
call eval_tree_evaluate (eval_tree)
print *
call eval_tree_write (eval_tree)
call stream_final (stream)
call ifile_final (ifile)
call eval_tree_final (eval_tree)
call var_list_final (var_list)
deallocate (var_list)
end subroutine expressions_test4
```

## Chapter 6

# Physics Models

While in previous WHIZARD versions, the physics model was partially hard-coded and injected into the main code via include files, the current version stores the accessible models in module variables. This allows for maintaining different models concurrently. Accessing the model is possible via pointers to the module variables; this is used by flavor objects, for instance.

### 6.1 Model module

```
<models.f90>≡  
  <File header>  
  
  module models  
  
    use iso_c_binding !NODEP!  
  <Use kinds>  
    use kinds, only: i8, i32 !NODEP!  
    use kinds, only: c_default_float !NODEP!  
  <Use strings>  
    use limits, only: VERTEX_TABLE_SCALE_FACTOR !NODEP!  
  <Use file utils>  
    use md5  
    use os_interface  
    use hashes, only: hash  
    use diagnostics !NODEP!  
    use ifiles  
    use syntax_rules  
    use lexers  
    use parser  
    use pdg_arrays  
    use variables  
    use expressions  
  
  <Standard module head>  
  
  <Models: public>  
  
  <Models: parameters>
```

```

    <Models: types>

    <Models: interfaces>

    <Models: variables>

contains

    <Models: procedures>

end module models

```

### 6.1.1 Physics Parameters

A parameter has a name, a value. Derived parameters also have a definition in terms of other parameters, which is stored as an `eval_tree`. External parameters are set by an external program.

```

<Models: parameters>≡
    integer, parameter :: PAR_NONE = 0
    integer, parameter :: PAR_INDEPENDENT = 1, PAR_DERIVED = 2
    integer, parameter :: PAR_EXTERNAL = 3

<Models: public>≡
    public :: parameter_t

<Models: types>≡
    type :: parameter_t
        private
        integer :: type = PAR_NONE
        type(string_t) :: name
        real(default) :: value = 0
        type(eval_tree_t) :: eval_tree
    end type parameter_t

```

Initialization depends on parameter type. Independent parameters are initialized by a constant value or a constant numerical expression (which may contain a unit). Derived parameters are initialized by an arbitrary numerical expression, which makes use of the current variable list. The expression is evaluated by the function `parameter_reset`.

```

<Models: procedures>≡
    subroutine parameter_init_independent_value (par, name, value)
        type(parameter_t), intent(out) :: par
        type(string_t), intent(in) :: name
        real(default), intent(in) :: value
        par%type = PAR_INDEPENDENT
        par%name = name
        par%value = value
    end subroutine parameter_init_independent_value

    subroutine parameter_init_independent (par, name, pn)
        type(parameter_t), intent(out) :: par

```

```

    type(string_t), intent(in) :: name
    type(parse_node_t), intent(in), target :: pn
    par%type = PAR_INDEPENDENT
    par%name = name
    call eval_tree_init_numeric_value (par%eval_tree, pn)
    par%value = eval_tree_get_real (par%eval_tree)
end subroutine parameter_init_independent

subroutine parameter_init_derived (par, name, pn, var_list)
    type(parameter_t), intent(out) :: par
    type(string_t), intent(in) :: name
    type(parse_node_t), intent(in), target :: pn
    type(var_list_t), intent(in), target :: var_list
    par%type = PAR_DERIVED
    par%name = name
    call eval_tree_init_expr (par%eval_tree, pn, var_list=var_list)
    call parameter_reset_derived (par)
end subroutine parameter_init_derived

subroutine parameter_init_external (par, name)
    type(parameter_t), intent(out) :: par
    type(string_t), intent(in) :: name
    par%type = PAR_EXTERNAL
    par%name = name
end subroutine parameter_init_external

```

The finalizer is needed for the evaluation tree in the definition.

```

<Models: procedures>+≡
subroutine parameter_final (par)
    type(parameter_t), intent(inout) :: par
    call eval_tree_final (par%eval_tree)
end subroutine parameter_final

```

All derived parameters should be recalculated if some independent parameters have changed:

```

<Models: procedures>+≡
subroutine parameter_reset_derived (par)
    type(parameter_t), intent(inout) :: par
    select case (par%type)
    case (PAR_DERIVED)
        call eval_tree_evaluate (par%eval_tree)
        par%value = eval_tree_get_real (par%eval_tree)
    end select
end subroutine parameter_reset_derived

```

Direct access to the parameter value:

```

<Models: procedures>+≡
function parameter_get_value_ptr (par) result (val)
    real(default), pointer :: val
    type(parameter_t), intent(in), target :: par
    val => par%value
end function parameter_get_value_ptr

```

Output. [We should have a formula format for the eval tree, suitable for input and output!]

*<Models: procedures>+≡*

```
subroutine parameter_write (par, unit, write_defs)
  type(parameter_t), intent(in) :: par
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: write_defs
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(3x,A)", advance="no") "parameter"
  write (u, "(1x,A,1x,A)", advance="no") char (par%name), "="
  write (u, "(G17.10)", advance="no") par%value
  select case (par%type)
  case (PAR_DERIVED)
    write (u, *) " ! derived"
    if (present (write_defs)) then
      if (write_defs) then
        call eval_tree_write (par%eval_tree, unit)
      end if
    end if
  case (PAR_EXTERNAL)
    write (u, *) " ! external"
  end select
end subroutine parameter_write
```

### 6.1.2 Particle codes

Let us define a few particle codes independent of the model.

SM fermions:

*<Models: types>+≡*

```
integer, parameter, public :: ELECTRON = 11
```

Gauge bosons:

*<Models: types>+≡*

```
integer, parameter, public :: GLUON = 21
integer, parameter, public :: PHOTON = 22
integer, parameter, public :: Z_BOSON = 23
integer, parameter, public :: W_BOSON = 24
```

Hadrons:

*<Models: types>+≡*

```
integer, parameter, public :: PROTON = 2212
```

*<Models: types>+≡*

```
integer, parameter, public :: PION = 111
integer, parameter, public :: PIPLUS = 211
integer, parameter, public :: PIMINUS = - PIPLUS
```



Hadron remnants (internal)

```
(Models: types)+≡  
    integer, parameter, public :: HADRON_REMNANT = 90  
    integer, parameter, public :: HADRON_REMNANT_SINGLET = 91  
    integer, parameter, public :: HADRON_REMNANT_TRIPLET = 92  
    integer, parameter, public :: HADRON_REMNANT_OCTET = 93
```

Generic particles for internal use in event analysis:

```
(Models: types)+≡  
    integer, parameter, public :: PRT_ANY = 81  
    integer, parameter, public :: PRT_VISIBLE = 82  
    integer, parameter, public :: PRT_CHARGED = 83  
    integer, parameter, public :: PRT_COLORED = 84
```

Further particle codes for internal use:

```
(Models: types)+≡  
    integer, parameter, public :: INVALID = 97  
    integer, parameter, public :: KEYSTONE = 98  
    integer, parameter, public :: COMPOSITE = 99
```

### 6.1.3 Spin codes

Somewhat redundant, but for better readability we define named constants for spin types. If the mass is nonzero, this is equal to the number of degrees of freedom.

```
(Models: types)+≡  
    integer, parameter, public :: UNKNOWN=0  
    integer, parameter, public :: SCALAR=1, SPINOR=2, VECTOR=3, &  
        VECTORSPINOR=4, TENSOR=5
```

Isospin types and charge types are counted in an analogous way, where charge type 1 is charge 0, 2 is charge 1/3, and so on. Zero always means unknown. Note that charge and isospin types have an explicit sign.

Color types are defined as the dimension of the representation.

### 6.1.4 Particle data

The particle-data type holds all information that pertains to a particular particle within a particular model. Information such as spin type, particle code etc. is stored within the object itself, while mass and width are associated to parameters, otherwise assumed zero.

```
(Models: public)+≡  
    public :: particle_data_t  
  
(Models: types)+≡  
    type :: particle_data_t  
        private  
        type(string_t) :: longname  
        integer :: pdg = UNDEFINED  
        logical :: is_visible = .true.  
        logical :: is_parton = .false.
```

```

logical :: is_gauge = .false.
logical :: is_left_handed = .false.
logical :: is_right_handed = .false.
logical :: has_antiparticle = .false.
logical :: p_is_stable = .true.
logical :: p_decays_isotropically = .false.
logical :: p_decays_diagonal = .false.
logical :: a_is_stable = .true.
logical :: a_decays_isotropically = .false.
logical :: a_decays_diagonal = .false.
logical :: p_polarized = .false.
logical :: a_polarized = .false.
type(string_t), dimension(:), allocatable :: name, anti
type(string_t) :: tex_name, tex_anti
integer :: spin_type = UNDEFINED
integer :: isospin_type = 1
integer :: charge_type = 1
integer :: color_type = 1
real(default), pointer :: mass_val => null ()
type(parameter_t), pointer :: mass_src => null ()
real(default), pointer :: width_val => null ()
type(parameter_t), pointer :: width_src => null ()
integer :: multiplicity = 1
end type particle_data_t

```

Initialize particle data with PDG long name and PDG code.  $\text{\TeX}$  names should be initialized to avoid issues with accessing unallocated string contents.

*(Models: public)*+≡

```
public :: particle_data_init
```

*(Models: procedures)*+≡

```

subroutine particle_data_init (prt, longname, pdg)
  type(particle_data_t), intent(out) :: prt
  type(string_t), intent(in) :: longname
  integer, intent(in) :: pdg
  prt%longname = longname
  prt%pdg = pdg
  prt%tex_name = ""
  prt%tex_anti = ""
end subroutine particle_data_init

```

Copy quantum numbers from another particle

*(Models: procedures)*+≡

```

subroutine particle_data_copy (prt, prt_src)
  type(particle_data_t), intent(inout) :: prt
  type(particle_data_t), intent(in) :: prt_src
  prt%is_visible = prt_src%is_visible
  prt%is_parton = prt_src%is_parton
  prt%is_gauge = prt_src%is_gauge
  prt%is_left_handed = prt_src%is_left_handed
  prt%is_right_handed = prt_src%is_right_handed
  prt%spin_type = prt_src%spin_type
  prt%isospin_type = prt_src%isospin_type

```

```

prt%charge_type = prt_src%charge_type
prt%color_type = prt_src%color_type
call particle_data_set_multiplicity (prt)
end subroutine particle_data_copy

```

Set particle quantum numbers.

*<Models: public>+≡*

```
public :: particle_data_set
```

*<Models: procedures>+≡*

```

subroutine particle_data_set (prt, &
    is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
    p_is_stable, p_decays_isotropically, p_decays_diagonal, &
    a_is_stable, a_decays_isotropically, a_decays_diagonal, &
    p_polarized, a_polarized, &
    name, anti, tex_name, tex_anti, &
    spin_type, isospin_type, charge_type, color_type, &
    mass_src, width_src)
type(particle_data_t), intent(inout) :: prt
logical, intent(in), optional :: is_visible, is_parton, is_gauge
logical, intent(in), optional :: is_left_handed, is_right_handed
logical, intent(in), optional :: p_is_stable
logical, intent(in), optional :: p_decays_isotropically, p_decays_diagonal
logical, intent(in), optional :: a_is_stable
logical, intent(in), optional :: a_decays_isotropically, a_decays_diagonal
logical, intent(in), optional :: p_polarized, a_polarized
type(string_t), dimension(:), intent(in), optional :: name, anti
type(string_t), intent(in), optional :: tex_name, tex_anti
integer, intent(in), optional :: spin_type, isospin_type
integer, intent(in), optional :: charge_type, color_type
type(parameter_t), intent(in), target, optional :: mass_src, width_src
if (present (is_visible)) prt%is_visible = is_visible
if (present (is_parton)) prt%is_parton = is_parton
if (present (is_gauge)) prt%is_gauge = is_gauge
if (present (is_left_handed)) prt%is_left_handed = is_left_handed
if (present (is_right_handed)) prt%is_right_handed = is_right_handed
if (present (p_is_stable)) prt%p_is_stable = p_is_stable
if (present (p_decays_isotropically)) &
    prt%p_decays_isotropically = p_decays_isotropically
if (present (p_decays_diagonal)) &
    prt%p_decays_diagonal = p_decays_diagonal
if (present (a_is_stable)) prt%a_is_stable = a_is_stable
if (present (a_decays_isotropically)) &
    prt%a_decays_isotropically = a_decays_isotropically
if (present (a_decays_diagonal)) &
    prt%a_decays_diagonal = a_decays_diagonal
if (present (p_polarized)) prt%p_polarized = p_polarized
if (present (a_polarized)) prt%a_polarized = a_polarized
if (present (name)) then
    allocate (prt%name (size (name)))
    prt%name = name
end if
if (present (anti)) then
    allocate (prt%anti (size (anti)))

```

```

        prt%anti = anti
        prt%has_antiparticle = .true.
    end if
    if (present (tex_name)) prt%tex_name = tex_name
    if (present (tex_anti)) prt%tex_anti = tex_anti
    if (present (spin_type)) prt%spin_type = spin_type
    if (present (isospin_type)) prt%isospin_type = isospin_type
    if (present (charge_type)) prt%charge_type = charge_type
    if (present (color_type)) prt%color_type = color_type
    if (present (mass_src)) then
        prt%mass_src => mass_src
        prt%mass_val => parameter_get_value_ptr (mass_src)
    end if
    if (present (width_src)) then
        prt%width_src => width_src
        prt%width_val => parameter_get_value_ptr (width_src)
    end if
    if (present (spin_type) .or. present (mass_src)) then
        call particle_data_set_multiplicity (prt)
    end if
end subroutine particle_data_set

```

Calculate the multiplicity given spin type and mass.

*(Models: procedures)*+≡

```

subroutine particle_data_set_multiplicity (prt)
    type(particle_data_t), intent(inout) :: prt
    if (prt%spin_type /= SCALAR) then
        if (associated (prt%mass_src)) then
            prt%multiplicity = prt%spin_type
        else if (prt%is_left_handed .or. prt%is_right_handed) then
            prt%multiplicity = 1
        else
            prt%multiplicity = 2
        end if
    end if
end subroutine particle_data_set_multiplicity

```

Set the mass/width value (not the pointer). The mass/width pointer must be allocated.

*(Models: procedures)*+≡

```

subroutine particle_data_set_mass (prt, mass)
    type(particle_data_t), intent(inout) :: prt
    real(default), intent(in) :: mass
    if (associated (prt%mass_val)) prt%mass_val = mass
end subroutine particle_data_set_mass

subroutine particle_data_set_width (prt, width)
    type(particle_data_t), intent(inout) :: prt
    real(default), intent(in) :: width
    if (associated (prt%width_val)) prt%width_val = width
end subroutine particle_data_set_width

```

Loose ends

*<Models: public>+≡*

```
public :: particle_data_freeze
```

*<Models: procedures>+≡*

```
subroutine particle_data_freeze (prt)
  type(particle_data_t), intent(inout) :: prt
  if (.not. allocated (prt%name)) allocate (prt%name (0))
  if (.not. allocated (prt%anti)) allocate (prt%anti (0))
end subroutine particle_data_freeze
```

Output

*<Models: procedures>+≡*

```
subroutine particle_data_write (prt, unit)
  type(particle_data_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, "(3x,A,1x,A)", advance="no") "particle", char (prt%longname)
  write (u, "(1x,I7)", advance="no") prt%pdg
  if (.not. prt%is_visible) write (u, "(3x,A)", advance="no") "invisible"
  if (prt%is_parton) write (u, "(3x,A)", advance="no") "parton"
  if (prt%is_gauge) write (u, "(3x,A)", advance="no") "gauge"
  if (prt%is_left_handed) write (u, "(3x,A)", advance="no") "left"
  if (prt%is_right_handed) write (u, "(3x,A)", advance="no") "right"
  write (u, *)
  write (u, "(5x,A)", advance="no") "name"
  if (allocated (prt%name)) then
    do i = 1, size (prt%name)
      write (u, "(1x,A)", advance="no") ' ' // char (prt%name(i)) // ' '
    end do
    write (u, *)
    if (prt%has_antiparticle) then
      write (u, "(5x,A)", advance="no") "anti"
      do i = 1, size (prt%anti)
        write (u, "(1x,A)", advance="no") ' ' // char (prt%anti(i)) // ' '
      end do
      write (u, *)
    end if
    if (prt%tex_name /= "") then
      write (u, "(5x,A)") &
        "tex_name " // ' ' // char (prt%tex_name) // ' '
    end if
    if (prt%has_antiparticle .and. prt%tex_anti /= "") then
      write (u, "(5x,A)") &
        "tex_anti " // ' ' // char (prt%tex_anti) // ' '
    end if
  else
    write (u, "(A)") "???"
  end if
  write (u, "(5x,A)", advance="no") "spin "
  select case (mod (prt%spin_type - 1, 2))
  case (0); write (u, "(I1)", advance="no") (prt%spin_type-1) / 2
  case default; write (u, "(I1,A)", advance="no") prt%spin_type-1, "/2"
```

```

end select
! write (u, "(3x,A,I1,A)") "! [multiplicity = ", prt%multiplicity, "]"
if (abs (prt%isospin_type) /= 1) then
  write (u, "(5x,A)", advance="no") "isospin "
  select case (mod (abs (prt%isospin_type) - 1, 2))
  case (0); write (u, "(I2)", advance="no") &
    sign (abs (prt%isospin_type) - 1, prt%isospin_type) / 2
  case default; write (u, "(I2,A)", advance="no") &
    sign (abs (prt%isospin_type) - 1, prt%isospin_type), "/2"
  end select
end if
if (abs (prt%charge_type) /= 1) then
  write (u, "(5x,A)", advance="no") "charge "
  select case (mod (abs (prt%charge_type) - 1, 3))
  case (0); write (u, "(I2)", advance="no") &
    sign (abs (prt%charge_type) - 1, prt%charge_type) / 3
  case default; write (u, "(I2,A)", advance="no") &
    sign (abs (prt%charge_type) - 1, prt%charge_type), "/3"
  end select
end if
if (prt%color_type /= 1) then
  write (u, "(5x,A,I2)", advance="no") "color ", prt%color_type
end if
write (u, *)
if (associated (prt%mass_src)) then
  write (u, "(5x,A)") "mass " // char (prt%mass_src%name)
  if (associated (prt%width_src)) then
    write (u, "(5x,A)") "width " // char (prt%width_src%name)
  end if
end if
end subroutine particle_data_write

```

Retrieve data:

*<Models: public>+≡*

```

public :: particle_data_get_pdg
public :: particle_data_get_pdg_anti

```

*<Models: procedures>+≡*

```

elemental function particle_data_get_pdg (prt) result (pdg)
  integer :: pdg
  type(particle_data_t), intent(in) :: prt
  pdg = prt%pdg
end function particle_data_get_pdg

elemental function particle_data_get_pdg_anti (prt) result (pdg)
  integer :: pdg
  type(particle_data_t), intent(in) :: prt
  if (prt%has_antiparticle) then
    pdg = - prt%pdg
  else
    pdg = prt%pdg
  end if
end function particle_data_get_pdg_anti

```

Predicates:

$\langle Models: public \rangle + \equiv$

```
public :: particle_data_is_visible
public :: particle_data_is_parton
public :: particle_data_is_gauge
public :: particle_data_is_left_handed
public :: particle_data_is_right_handed
public :: particle_data_has_antiparticle
public :: particle_data_is_stable
public :: particle_data_decays_isotropically
public :: particle_data_decays_diagonal
public :: particle_data_is_polarized
```

$\langle Models: procedures \rangle + \equiv$

```
elemental function particle_data_is_visible (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_visible
end function particle_data_is_visible

elemental function particle_data_is_parton (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_parton
end function particle_data_is_parton

elemental function particle_data_is_gauge (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_gauge
end function particle_data_is_gauge

elemental function particle_data_is_left_handed (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_left_handed
end function particle_data_is_left_handed

elemental function particle_data_is_right_handed (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_right_handed
end function particle_data_is_right_handed

elemental function particle_data_has_antiparticle (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%has_antiparticle
end function particle_data_has_antiparticle

elemental function particle_data_is_stable (prt, anti) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
```

```

        if (anti) then
            flag = prt%a_is_stable
        else
            flag = prt%p_is_stable
        end if
    else
        flag = prt%p_is_stable
    end if
end function particle_data_is_stable

elemental function particle_data_decays_isotropically &
    (prt, anti) result (flag)
    logical :: flag
    type(particle_data_t), intent(in) :: prt
    logical, intent(in), optional :: anti
    if (present (anti)) then
        if (anti) then
            flag = prt%a_decays_isotropically
        else
            flag = prt%p_decays_isotropically
        end if
    else
        flag = prt%p_decays_isotropically
    end if
end function particle_data_decays_isotropically

elemental function particle_data_decays_diagonal &
    (prt, anti) result (flag)
    logical :: flag
    type(particle_data_t), intent(in) :: prt
    logical, intent(in), optional :: anti
    if (present (anti)) then
        if (anti) then
            flag = prt%a_decays_diagonal
        else
            flag = prt%p_decays_diagonal
        end if
    else
        flag = prt%p_decays_diagonal
    end if
end function particle_data_decays_diagonal

elemental function particle_data_is_polarized (prt, anti) result (flag)
    logical :: flag
    type(particle_data_t), intent(in) :: prt
    logical, intent(in), optional :: anti
    logical :: a
    if (present (anti)) then
        a = anti
    else
        a = .false.
    end if
    if (a) then
        flag = prt%a_polarized
    end if
end function particle_data_is_polarized

```



```

else
    flag = prt%p_polarized
end if
end function particle_data_is_polarized

```

Names. Return the first name in the list (or the first antiparticle name)

```

⟨Models: public⟩+≡
    public :: particle_data_get_name

⟨Models: procedures⟩+≡
    elemental function particle_data_get_name &
        (prt, is_antiparticle) result (name)
        type(string_t) :: name
        type(particle_data_t), intent(in) :: prt
        logical, intent(in) :: is_antiparticle
        name = "???"
        if (is_antiparticle) then
            if (prt%has_antiparticle) then
                if (allocated (prt%anti)) then
                    if (size(prt%anti) > 0) name = prt%anti(1)
                end if
            else
                if (allocated (prt%name)) then
                    if (size (prt%name) > 0) name = prt%name(1)
                end if
            end if
        else
            if (allocated (prt%name)) then
                if (size (prt%name) > 0) name = prt%name(1)
            end if
        end if
    end function particle_data_get_name

```

Same for the T<sub>E</sub>X name.

```

⟨Models: public⟩+≡
    public :: particle_data_get_tex_name

⟨Models: procedures⟩+≡
    elemental function particle_data_get_tex_name &
        (prt, is_antiparticle) result (name)
        type(string_t) :: name
        type(particle_data_t), intent(in) :: prt
        logical, intent(in) :: is_antiparticle
        if (is_antiparticle) then
            if (prt%has_antiparticle) then
                name = prt%tex_anti
            else
                name = prt%tex_name
            end if
        else
            name = prt%tex_name
        end if
        if (name == "") name = particle_data_get_name (prt, is_antiparticle)
    end function particle_data_get_tex_name

```

Quantum numbers

*<Models: public>+≡*

```
public :: particle_data_get_spin_type
public :: particle_data_get_multiplicity
public :: particle_data_get_isospin_type
public :: particle_data_get_charge_type
public :: particle_data_get_color_type
```

*<Models: procedures>+≡*

```
elemental function particle_data_get_spin_type (prt) result (type)
  integer :: type
  type(particle_data_t), intent(in) :: prt
  type = prt%spin_type
end function particle_data_get_spin_type

elemental function particle_data_get_multiplicity (prt) result (type)
  integer :: type
  type(particle_data_t), intent(in) :: prt
  type = prt%multiplicity
end function particle_data_get_multiplicity

elemental function particle_data_get_isospin_type (prt) result (type)
  integer :: type
  type(particle_data_t), intent(in) :: prt
  type = prt%isospin_type
end function particle_data_get_isospin_type

elemental function particle_data_get_charge_type (prt) result (type)
  integer :: type
  type(particle_data_t), intent(in) :: prt
  type = prt%charge_type
end function particle_data_get_charge_type

elemental function particle_data_get_color_type (prt) result (type)
  integer :: type
  type(particle_data_t), intent(in) :: prt
  type = prt%color_type
end function particle_data_get_color_type
```

In the MSSM, neutralinos can have a negative mass. This is relevant for computing matrix elements. However, within the WHIZARD main program we are interested only in kinematics, therefore we return the absolute value of the particle mass. If desired, we can extract the sign separately.

*<Models: public>+≡*

```
public :: particle_data_get_charge
public :: particle_data_get_mass
public :: particle_data_get_mass_sign
public :: particle_data_get_width
public :: particle_data_get_isospin
```

*<Models: procedures>+≡*

```
elemental function particle_data_get_charge (prt) result (charge)
  real(default) :: charge
```

```

type(particle_data_t), intent(in) :: prt
if (prt%charge_type /= 0) then
    charge = real (sign ((abs(prt%charge_type) - 1), &
        prt%charge_type), default) / 3
else
    charge = 0
end if
end function particle_data_get_charge

elemental function particle_data_get_mass (prt) result (mass)
    real(default) :: mass
    type(particle_data_t), intent(in) :: prt
    if (associated (prt%mass_val)) then
        mass = abs (prt%mass_val)
    else
        mass = 0
    end if
end function particle_data_get_mass

elemental function particle_data_get_mass_sign (prt) result (sgn)
    integer :: sgn
    type(particle_data_t), intent(in) :: prt
    if (associated (prt%mass_val)) then
        sgn = sign (1._default, prt%mass_val)
    else
        sgn = 0
    end if
end function particle_data_get_mass_sign

elemental function particle_data_get_width (prt) result (width)
    real(default) :: width
    type(particle_data_t), intent(in) :: prt
    if (associated (prt%width_val)) then
        width = prt%width_val
    else
        width = 0
    end if
end function particle_data_get_width

elemental function particle_data_get_isospin (prt) result (isospin)
    real(default) :: isospin
    type(particle_data_t), intent(in) :: prt
    if (prt%isospin_type /= 0) then
        isospin = real (sign (abs(prt%isospin_type) - 1, &
            prt%isospin_type), default) / 2
    else
        isospin = 0
    end if
end function particle_data_get_isospin

```

Given an array of particles, return a PDG-array object that consists of all charged particles in the array. We need to explicitly add their antiparticles.

*(Models: procedures)*+≡

```

function particle_data_get_charged_pdg (prt) result (aval)
  type(pdg_array_t) :: aval, aval_p, aval_a
  type(particle_data_t), dimension(:), intent(in) :: prt
  aval_p = pack ( prt%pdg, abs (prt%charge_type) > 1)
  aval_a = pack (-prt%pdg, abs (prt%charge_type) > 1 &
    .and. prt%has_antiparticle )
  aval = aval_p // aval_a
end function particle_data_get_charged_pdg

```

The same for color.

*(Models: procedures)+≡*

```

function particle_data_get_colored_pdg (prt) result (aval)
  type(pdg_array_t) :: aval, aval_p, aval_a
  type(particle_data_t), dimension(:), intent(in) :: prt
  aval_p = pack ( prt%pdg, abs (prt%color_type) > 1)
  aval_a = pack (-prt%pdg, abs (prt%color_type) > 1 &
    .and. prt%has_antiparticle )
  aval = aval_p // aval_a
end function particle_data_get_colored_pdg

```

### 6.1.5 Vertex data

The vertex object contains an array of particle-data pointers, for which we need a separate type. (We could use the flavor type defined in another module.)

The program does not (yet?) make use of vertex definitions, so they are not stored here.

*(Models: types)+≡*

```

type :: particle_p
  type(particle_data_t), pointer :: p => null ()
end type particle_p

```

*(Models: types)+≡*

```

type :: vertex_t
  logical :: trilinear
  integer, dimension(:), allocatable :: pdg
  type(particle_p), dimension(:), allocatable :: prt
end type vertex_t

```

Initialize using PDG codes. The model is used for finding particle data pointers associated with the pdg codes.

*(Models: procedures)+≡*

```

subroutine vertex_init (vtx, pdg, model)
  type(vertex_t), intent(out) :: vtx
  integer, dimension(:), intent(in) :: pdg
  type(model_t), intent(in), target, optional :: model
  integer :: i
  allocate (vtx%pdg (size (pdg)))
  allocate (vtx%prt (size (pdg)))
  vtx%trilinear = size (pdg) == 3
  vtx%pdg = pdg

```

```

    if (present (model)) then
      do i = 1, size (pdg)
        vtx%prt(i)%p => model_get_particle_ptr (model, pdg(i))
      end do
    end if
  end subroutine vertex_init

```

*<Models: procedures>+≡*

```

subroutine vertex_write (vtx, unit)
  type(vertex_t), intent(in) :: vtx
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, "(3x,A)", advance="no") "vertex"
  do i = 1, size (vtx%prt)
    if (associated (vtx%prt(i)%p)) then
      write (u, "(1x,A)", advance="no") &
        '"" // char (particle_data_get_name &
          (vtx%prt(i)%p, vtx%pdg(i) < 0)) &
        // '""
    else
      write (u, "(1x,I7)", advance="no") vtx%pdg(i)
    end if
  end do
  write (u, *)
end subroutine vertex_write

```

### 6.1.6 Vertex lookup table

The vertex lookup table is a hash table: given two particle codes, we check which codes are allowed for the third one.

The size of the hash table should be large enough that collisions are rare. We first select a size based on the number of vertices (multiplied by six because all permutations count), with some margin, and then choose the smallest integer power of two larger than this.

*<Limits: public parameters>+≡*

```

integer, parameter, public :: VERTEX_TABLE_SCALE_FACTOR = 60

```

*<Models: procedures>+≡*

```

function vertex_table_size (n_vtx) result (n)
  integer(i32) :: n
  integer, intent(in) :: n_vtx
  integer :: i, s
  s = VERTEX_TABLE_SCALE_FACTOR * n_vtx
  n = 1
  do i = 1, 31
    n = ishft (n, 1)
    s = ishft (s, -1)
    if (s == 0) exit
  end do
end function vertex_table_size

```

The specific hash function takes two particle codes (arbitrary integers) and returns a 32-bit integer. It makes use of the universal function `hash` which operates on a byte array.

```

<Models: procedures>+≡
  function hash2 (pdg1, pdg2)
    integer(i32) :: hash2
    integer, intent(in) :: pdg1, pdg2
    integer(i8), dimension(1) :: mold
    hash2 = hash (transfer ((/pdg1, pdg2/), mold))
  end function hash2

```

Each entry in the vertex table stores the two particle codes and an array of possibilities for the third code.

```

<Models: types>+≡
  type :: vertex_table_entry_t
    integer :: pdg1 = 0, pdg2 = 0
    integer :: n = 0
    integer, dimension(:), allocatable :: pdg3
  end type vertex_table_entry_t

```

The vertex table:

```

<Models: types>+≡
  type :: vertex_table_t
    type(vertex_table_entry_t), dimension(:), allocatable :: entry
    integer :: n_collisions = 0
    integer(i32) :: mask
  end type vertex_table_t

```

Initializing the vertex table: This is done in two passes. First, we scan all permutations for all vertices and count the number of entries in each bucket of the hashtable. Then, the buckets are allocated accordingly and filled.

Collision resolution is done by just incrementing the hash value until an empty bucket is found. The vertex table size is fixed, since we know from the beginning the number of entries.

```

<Models: procedures>+≡
  subroutine vertex_table_init (vt, prt, vtx)
    type(vertex_table_t), intent(out) :: vt
    type(particle_data_t), dimension(:), intent(in) :: prt
    type(vertex_t), dimension(:), intent(in) :: vtx
    integer :: n_prt, n_vtx, vt_size, i, p1, p2, p3
    integer, dimension(3) :: p
    n_prt = size (prt)
    n_vtx = size (vtx)
    vt_size = vertex_table_size (count (vtx%trilinear))
    vt%mask = vt_size - 1
    allocate (vt%entry (0:vt_size-1))
    do i = 1, n_vtx
      if (vtx(i)%trilinear) then
        p = vtx(i)%pdg
        p1 = p(1); p2 = p(2)
        call create (hash2 (p1, p2))
      end if
    end do
  end subroutine vertex_table_init

```

```

        if (p(2) /= p(3)) then
            p2 = p(3)
            call create (hash2 (p1, p2))
        end if
        if (p(1) /= p(2)) then
            p1 = p(2); p2 = p(1)
            call create (hash2 (p1, p2))
            if (p(1) /= p(3)) then
                p2 = p(3)
                call create (hash2 (p1, p2))
            end if
        end if
        if (p(1) /= p(3)) then
            p1 = p(3); p2 = p(1)
            call create (hash2 (p1, p2))
            if (p(1) /= p(2)) then
                p2 = p(2)
                call create (hash2 (p1, p2))
            end if
        end if
    end if
end do
do i = 0, vt_size - 1
    allocate (vt%entry(i)%pdg3 (vt%entry(i)%n))
end do
vt%entry%n = 0
do i = 1, n_vtx
    if (vtx(i)%trilinear) then
        p = vtx(i)%pdg
        p1 = p(1); p2 = p(2); p3 = p(3)
        call register (hash2 (p1, p2))
        if (p(2) /= p(3)) then
            p2 = p(3); p3 = p(2)
            call register (hash2 (p1, p2))
        end if
        if (p(1) /= p(2)) then
            p1 = p(2); p2 = p(1); p3 = p(3)
            call register (hash2 (p1, p2))
            if (p(1) /= p(3)) then
                p2 = p(3); p3 = p(1)
                call register (hash2 (p1, p2))
            end if
        end if
        if (p(1) /= p(3)) then
            p1 = p(3); p2 = p(1); p3 = p(2)
            call register (hash2 (p1, p2))
            if (p(1) /= p(2)) then
                p2 = p(2); p3 = p(1)
                call register (hash2 (p1, p2))
            end if
        end if
    end if
end do
contains

```

```

recursive subroutine create (hashval)
  integer(i32), intent(in) :: hashval
  integer :: h
  h = iand (hashval, vt%mask)
  if (vt%entry(h)%n == 0) then
    vt%entry(h)%pdg1 = p1
    vt%entry(h)%pdg2 = p2
    vt%entry(h)%n = 1
  else if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
    vt%entry(h)%n = vt%entry(h)%n + 1
  else
    vt%n_collisions = vt%n_collisions + 1
    call create (hashval + 1)
  end if
end subroutine create
recursive subroutine register (hashval)
  integer(i32), intent(in) :: hashval
  integer :: h
  h = iand (hashval, vt%mask)
  if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
    vt%entry(h)%n = vt%entry(h)%n + 1
    vt%entry(h)%pdg3(vt%entry(h)%n) = p3
  else
    call register (hashval + 1)
  end if
end subroutine register
end subroutine vertex_table_init

```

Output

```

<Models: procedures>+≡
subroutine vertex_table_write (vt, unit)
  type(vertex_table_t), intent(in) :: vt
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, *) "vertex hash table:"
  write (u, *) "  size = ", size (vt%entry)
  write (u, *) "  used = ", count (vt%entry%n /= 0)
  write (u, *) "  coll = ", vt%n_collisions
  do i = lbound (vt%entry, 1), ubound (vt%entry, 1)
    if (vt%entry(i)%n /= 0) then
      write (u, *) "    ", i, ":", &
        vt%entry(i)%pdg1, vt%entry(i)%pdg2, "->", vt%entry(i)%pdg3
    end if
  end do
end subroutine vertex_table_write

```

Return the array of particle codes that match the given pair.

```

<Models: procedures>+≡
subroutine vertex_table_match (vt, pdg1, pdg2, pdg3)
  type(vertex_table_t), intent(in) :: vt
  integer, intent(in) :: pdg1, pdg2
  integer, dimension(:), allocatable, intent(out) :: pdg3

```



```

integer :: vt_size
vt_size = size (vt%entry)
call match (hash2 (pdg1, pdg2))
contains
recursive subroutine match (hashval)
integer(i32), intent(in) :: hashval
integer :: h
h = iand (hashval, vt%mask)
if (vt%entry(h)%n == 0) then
allocate (pdg3 (0))
else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
allocate (pdg3 (size (vt%entry(h)%pdg3)))
pdg3 = vt%entry(h)%pdg3
else
call match (hashval + 1)
end if
end subroutine match
end subroutine vertex_table_match

```

Return true if the triplet is represented as a vertex.

*(Models: procedures)+≡*

```

function vertex_table_check (vt, pdg1, pdg2, pdg3) result (flag)
type(vertex_table_t), intent(in) :: vt
integer, intent(in) :: pdg1, pdg2, pdg3
logical :: flag
integer :: vt_size
vt_size = size (vt%entry)
flag = check (hash2 (pdg1, pdg2))
contains
recursive function check (hashval) result (flag)
integer(i32), intent(in) :: hashval
integer :: h
logical :: flag
h = iand (hashval, vt%mask)
if (vt%entry(h)%n == 0) then
flag = .false.
else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
flag = any (vt%entry(h)%pdg3 == pdg3)
else
flag = check (hashval + 1)
end if
end function check
end function vertex_table_check

```

### 6.1.7 Model data

A model object holds all information about parameters, particles, and vertices. For models that require an external program for parameter calculation, there is the pointer to a function that does this calculation, given the set of independent and derived parameters.

*(Models: public)+≡*

```

public :: model_t

```

```

<Models: types>+=
  type :: model_t
    private
      type(string_t) :: name
      character(32) :: md5sum = ""
      type(parameter_t), dimension(:), allocatable :: par
      type(particle_data_t), dimension(:), allocatable :: prt
      type(vertex_t), dimension(:), allocatable :: vtx
      type(vertex_table_t) :: vt
      type(var_list_t) :: var_list
      type(string_t) :: dlname
      procedure(model_init_external_parameters), nopass, pointer :: &
        init_external_parameters => null ()
      type(dlaccess_t) :: dlaccess
    contains
      <Models: model: TBP>
    end type model_t

```

This is the interface for a procedure that initializes the calculation of external parameters, given the array of all parameters.

```

<Models: interfaces>=
  abstract interface
    subroutine model_init_external_parameters (par) bind (C)
      import
      real(c_default_float), dimension(*), intent(inout) :: par
    end subroutine model_init_external_parameters
  end interface

```

Initialization: Specify the number of parameters, particles, vertices and allocate memory. If an associated DL library is specified, load this library.

```

<Models: procedures>+=
  subroutine model_init &
    (model, name, libname, os_data, n_par, n_prt, n_vtx)
    type(model_t), intent(out) :: model
    type(string_t), intent(in) :: name, libname
    type(os_data_t), intent(in) :: os_data
    integer, intent(in) :: n_par, n_prt, n_vtx
    type(c_funptr) :: c_funptr
    type(string_t) :: libpath
    model%name = name
    allocate (model%par (n_par))
    allocate (model%prt (n_prt))
    allocate (model%vtx (n_vtx))
    if (libname /= "") then
      if (.not. os_data%use_testfiles) then
        libpath = os_data%whizard_models_libpath_local
        model%dlname = os_get_dlname ( &
          libpath // "/" // libname, os_data, ignore=.true.)
      end if
      if (model%dlname == "") then
        libpath = os_data%whizard_models_libpath
        model%dlname = os_get_dlname (libpath // "/" // libname, os_data)
      end if
    end if
  end subroutine model_init

```

```

else
  model%dlname = ""
end if
if (model%dlname /= "") then
  if (.not. dlaccess_is_open (model%dlaccess)) then
    if (logging) &
      call msg_message ("Loading model auxiliary library '" &
        // char (libpath) // "/" // char (model%dlname) // "'")
    call dlaccess_init (model%dlaccess, os_data%whizard_models_libpath, &
      model%dlname, os_data)
    if (dlaccess_has_error (model%dlaccess)) then
      call msg_message (char (dlaccess_get_error (model%dlaccess)))
      call msg_fatal ("Loading model auxiliary library '" &
        // char (model%dlname) // "' failed")
      return
    end if
    c_fptr = dlaccess_get_c_funptr (model%dlaccess, &
      var_str ("init_external_parameters"))
    if (dlaccess_has_error (model%dlaccess)) then
      call msg_message (char (dlaccess_get_error (model%dlaccess)))
      call msg_fatal ("Loading function from auxiliary library '" &
        // char (model%dlname) // "' failed")
      return
    end if
    call c_f_procpointer (c_fptr, model% init_external_parameters)
  end if
end if
end subroutine model_init

```

Finalization: The variable list is the only part that contains pointers.

```

<Models: public>+≡
  public :: model_final

<Models: procedures>+≡
  subroutine model_final (model)
    type(model_t), intent(inout) :: model
    call var_list_final (model%var_list)
    if (model%dlname /= "") call dlaccess_final (model%dlaccess)
  end subroutine model_final

```

Output. By default, the output is in the form of an input file. If `verbose` is true, for each derived parameter the definition (eval tree) is displayed, and the vertex hash table is shown.

```

<Models: public>+≡
  public :: model_write

<Models: model: TBP>≡
  procedure :: write => model_write

<Models: procedures>+≡
  subroutine model_write (model, unit, verbose)
    class(model_t), intent(in) :: model
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose

```

```

integer :: u, i
u = output_unit (unit); if (u < 0) return
write (u, *) 'model "', char (model%name), '"
write (u, *) '! md5sum = "', model%md5sum, '"
do i = 1, size (model%par)
    call parameter_write (model%par(i), unit, verbose)
    write (u, *)
end do
do i = 1, size (model%prt)
    call particle_data_write (model%prt(i), unit)
end do
do i = 1, size (model%vtx)
    call vertex_write (model%vtx(i), unit)
end do
if (present (verbose)) then
    if (verbose) then
        call vertex_table_write (model%vt, unit)
        call var_list_write (model%var_list, unit)
    end if
end if
end subroutine model_write

```

Accessing contents:

```

<Models: public>+≡
    public :: model_get_name

<Models: model: TBP>+≡
    procedure :: get_name => model_get_name

<Models: procedures>+≡
    function model_get_name (model) result (name)
        type(string_t) :: name
        class(model_t), intent(in) :: model
        name = model%name
    end function model_get_name

```

Retrieve the MD5 sum of a model (actually, of the model file).

```

<Models: public>+≡
    public :: model_get_md5sum

<Models: procedures>+≡
    function model_get_md5sum (model) result (md5sum)
        character(32) :: md5sum
        type(model_t), intent(in) :: model
        md5sum = model%md5sum
    end function model_get_md5sum

```

Retrieve a MD5 sum for the current model parameter values. This is done by writing them to a temporary file, using a standard format.

```

<Models: public>+≡
    public :: model_get_parameters_md5sum

```

```

<Models: procedures>+≡
function model_get_parameters_md5sum (model) result (par_md5sum)
  character(32) :: par_md5sum
  type(model_t), intent(in) :: model
  real(default), dimension(:), allocatable :: par
  integer :: unit
  call model_parameters_to_array (model, par)
  unit = free_unit ()
  open (unit, status="scratch", action="readwrite")
  write (unit, "(ES19.12)") par
  rewind (unit)
  par_md5sum = md5sum (unit)
  close (unit)
end function model_get_parameters_md5sum

```

Retrieve the MD5 sum for the current polarization setup (aka whether particles are to be treated as potentially polarized)

```

<Models: public>+≡
public :: model_get_polarized_md5sum

<Models: procedures>+≡
function model_get_polarized_md5sum (model) result (pol_md5sum)
  character(32) :: pol_md5sum
  type(model_t), intent(in) :: model
  integer :: unit, i
  unit = free_unit ()
  open (unit, status="scratch", action="readwrite")
  if (size (model%prt) > 0) then
    do i = 1, size (model%prt)
      write (unit, *) &
        char (particle_data_get_name (model%prt(i), .false.)), " ", &
        particle_data_is_polarized (model%prt(i)), " "
      if (particle_data_has_antiparticle (model%prt(i))) write (unit, *) &
        char (particle_data_get_name (model%prt(i), .true.)), " ", &
        particle_data_is_polarized (model%prt(i), .true.), " "
    end do
  end if
  rewind (unit)
  pol_md5sum = md5sum (unit)
  close (unit)
end function model_get_polarized_md5sum

```

Parameters are defined by an expression which may be constant or arbitrary.

```

<Models: interfaces>+≡
interface model_set_parameter
  module procedure model_set_parameter_constant
  module procedure model_set_parameter_parse_node
end interface

<Models: procedures>+≡
subroutine model_set_parameter_constant (model, i, name, value)
  type(model_t), intent(inout), target :: model
  integer, intent(in) :: i

```

```

    type(string_t), intent(in) :: name
    real(default), intent(in) :: value
    logical, save, target :: known = .true.
    call parameter_init_independent_value (model%par(i), name, value)
    call var_list_append_real_ptr &
        (model%var_list, name, parameter_get_value_ptr (model%par(i)), known, &
         intrinsic=.true.)
end subroutine model_set_parameter_constant

subroutine model_set_parameter_parse_node (model, i, name, pn, constant)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(string_t), intent(in) :: name
    type(parse_node_t), intent(in), target :: pn
    logical, intent(in) :: constant
    logical, save, target :: known = .true.
    if (constant) then
        call parameter_init_independent (model%par(i), name, pn)
    else
        call parameter_init_derived (model%par(i), name, pn, model%var_list)
    end if
    call var_list_append_real_ptr &
        (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
         is_known=known, locked=.not.constant, intrinsic=.true.)
end subroutine model_set_parameter_parse_node

subroutine model_set_parameter_external (model, i, name)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(string_t), intent(in) :: name
    logical, save, target :: known = .true.
    call parameter_init_external (model%par(i), name)
    call var_list_append_real_ptr &
        (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
         is_known=known, locked=.true., intrinsic=.true.)
end subroutine model_set_parameter_external

```

Return the pointer to a parameter

*(Models: procedures)*+≡

```

function model_get_parameter_ptr (model, par_name) result (par)
    type(parameter_t), pointer :: par
    type(model_t), intent(in), target :: model
    type(string_t), intent(in), optional :: par_name
    integer :: i
    par => null ()
    if (present (par_name)) then
        do i = 1, size (model%par)
            if (model%par(i)%name == par_name) then
                par => model%par(i); exit
            end if
        end do
    end if
    if (.not. associated (par)) then
        call msg_fatal (" Model '" // char (model%name) // "' // &

```

```

            " has no parameter '" // char (par_name) // '"')
        end if
    end if
end function model_get_parameter_ptr

```

Return the value of a particular parameter

```

<Models: public>+≡
    public :: model_get_parameter_value

<Models: procedures>+≡
    function model_get_parameter_value (model, par_name) result (val)
        real(default) :: val
        type(model_t), intent(in), target :: model
        type(string_t), intent(in), optional :: par_name
        val = parameter_get_value_ptr (model_get_parameter_ptr (model, par_name))
    end function model_get_parameter_value

```

Return the number of parameters

```

<Models: public>+≡
    public :: model_get_n_parameters

<Models: procedures>+≡
    function model_get_n_parameters (model) result (n)
        integer :: n
        type(model_t), intent(in) :: model
        n = size (model%par)
    end function model_get_n_parameters

```

Transform the parameters into a single real-valued array. The first version uses the Fortran default kind, the second one the C default kind. (Usually, the two will be identical).

```

<Models: public>+≡
    public :: model_parameters_to_array
    public :: model_parameters_to_c_array

<Models: procedures>+≡
    subroutine model_parameters_to_array (model, array)
        type(model_t), intent(in) :: model
        real(default), dimension(:), allocatable :: array
        integer :: i
        if (allocated (array)) deallocate (array)
        allocate (array (size (model%par)))
        do i = 1, size (model%par)
            array(i) = model%par(i)%value
        end do
    end subroutine model_parameters_to_array

    subroutine model_parameters_to_c_array (model, array)
        type(model_t), intent(in) :: model
        real(c_default_float), dimension(:), allocatable :: array
        allocate (array (size (model%par)))
        array = model%par%value
    end subroutine model_parameters_to_c_array

```

```

subroutine model_parameters_from_c_array (model, array)
  type(model_t), intent(inout) :: model
  real(c_default_float), dimension(:), intent(in) :: array
  if (size (array) == size (model%par)) then
    model%par%value = array
  else
    call msg_bug ("Model '" // char (model%name) // "': size mismatch " &
      // "in parameter array")
  end if
end subroutine model_parameters_from_c_array

```

Rcalculate all derived parameters.

*<Models: public>+≡*

```

public :: model_parameters_update

```

*<Models: procedures>+≡*

```

subroutine model_parameters_update (model)
  type(model_t), intent(inout) :: model
  integer :: i
  real(default), dimension(:), allocatable :: par
  do i = 1, size (model%par)
    call parameter_reset_derived (model%par(i))
  end do
  if (associated (model% init_external_parameters)) then
    call model_parameters_to_c_array (model, par)
    call model% init_external_parameters (par)
    call model_parameters_from_c_array (model, par)
  end if
end subroutine model_parameters_update

```

Initialize particle data with PDG long name and PDG code.

*<Models: procedures>+≡*

```

subroutine model_init_particle (model, i, longname, pdg)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: longname
  integer, intent(in) :: pdg
  type(pdg_array_t) :: aval
  call particle_data_init (model%prt(i), longname, pdg)
  aval = pdg
  call var_list_append_pdg_array &
    (model%var_list, longname, aval, locked=.true., intrinsic=.true.)
end subroutine model_init_particle

```

Copy quantum numbers from another particle

*<Models: procedures>+≡*

```

subroutine model_copy_particle_data (model, i, name_src)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name_src
  call particle_data_copy (model%prt(i), &
    model_get_particle_ptr (model, &
      model_get_particle_pdg (model, name_src)))

```



```
end subroutine model_copy_particle_data
```

Set particle quantum numbers individually. Names get a \* prepended.

*(Models: procedures)*+≡

```
subroutine model_set_particle_data (model, i, &
    is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
    name, anti, tex_name, tex_anti, &
    spin_type, isospin_type, charge_type, color_type, &
    mass_src, width_src)
    type(model_t), intent(inout) :: model
    integer, intent(in) :: i
    logical, intent(in), optional :: is_visible, is_parton, is_gauge
    logical, intent(in), optional :: is_left_handed, is_right_handed
    type(string_t), dimension(:), intent(in), optional :: name, anti
    type(string_t), intent(in), optional :: tex_name, tex_anti
    integer, intent(in), optional :: spin_type, isospin_type
    integer, intent(in), optional :: charge_type, color_type
    type(parameter_t), intent(in), optional, target :: mass_src, width_src
    integer :: j
    type(pdg_array_t) :: aval
    logical, parameter :: is_stable = .true.
    logical, parameter :: decays_isotropically = .false.
    logical, parameter :: decays_diagonal = .false.
    logical, parameter :: p_polarized = .false.
    logical, parameter :: a_polarized = .false.
    call particle_data_set (model%prt(i), &
        is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
        is_stable, decays_isotropically, decays_diagonal, &
        is_stable, decays_isotropically, decays_diagonal, &
        p_polarized, a_polarized, &
        name, anti, tex_name, tex_anti, &
        spin_type, isospin_type, charge_type, color_type, &
        mass_src, width_src)
    if (present (name)) then
        aval = particle_data_get_pdg (model%prt(i))
        do j = 1, size (name)
            call var_list_append_pdg_array &
                (model%var_list, name(j), aval, locked=.true., intrinsic=.true.)
        end do
    end if
    if (present (anti)) then
        aval = - particle_data_get_pdg (model%prt(i))
        do j = 1, size (anti)
            call var_list_append_pdg_array &
                (model%var_list, anti(j), aval, locked=.true., intrinsic=.true.)
        end do
    end if
end subroutine model_set_particle_data
```

*(Models: procedures)*+≡

```
subroutine model_freeze_particle_data (model, i)
    type(model_t), intent(inout) :: model
    integer, intent(in) :: i
```

```

        call particle_data_freeze (model%prt(i))
    end subroutine model_freeze_particle_data

```

Return a pointer to the particle-data object that belongs to the specified PDG code or name.

*<Models: public>+≡*

```

    public :: model_get_particle_ptr

```

*<Models: procedures>+≡*

```

function model_get_particle_ptr (model, pdg) result (prt)
    type(particle_data_t), pointer :: prt
    type(model_t), intent(in), target :: model
    integer, intent(in) :: pdg
    integer :: i
    prt => null ()
    if (pdg /= UNDEFINED) then
        do i = 1, size (model%prt)
            if (model%prt(i)%pdg == abs (pdg)) then
                prt => model%prt(i); exit
            end if
        end do
        if (.not. associated (prt)) then
            write (msg_buffer, "(1x,A,1x,I0)") "PDG code =", pdg
            call msg_message
            call msg_fatal (" Model ' " // char (model%name) // "' " // &
                " has no particle with this PDG code")
        end if
    end if
end function model_get_particle_ptr

```

Test if a particle with the given PDG code exists in the model.

*<Models: public>+≡*

```

    public :: model_test_particle

```

*<Models: procedures>+≡*

```

function model_test_particle (model, pdg) result (exists)
    logical :: exists
    type(particle_data_t), pointer :: prt
    type(model_t), intent(in), target :: model
    integer, intent(in) :: pdg
    integer :: i
    prt => null ()
    if (pdg /= UNDEFINED) then
        do i = 1, size (model%prt)
            if (model%prt(i)%pdg == abs (pdg)) then
                prt => model%prt(i); exit
            end if
        end do
        exists = associated(prt)
    else
        exists = .false.
    end if
end function model_test_particle

```

Set the value, not the pointer. If the PDG code is not valid, do nothing.

```

<Models: public>+≡
    public :: model_set_particle_mass
    public :: model_set_particle_width

<Models: procedures>+≡
    subroutine model_set_particle_mass (model, pdg, mass)
        type(model_t), intent(inout) :: model
        integer, intent(in) :: pdg
        real(default), intent(in) :: mass
        type(particle_data_t), pointer :: prt
        prt => model_get_particle_ptr (model, pdg)
        if (associated (prt)) call particle_data_set_mass (prt, mass)
    end subroutine model_set_particle_mass

    subroutine model_set_particle_width (model, pdg, width)
        type(model_t), intent(inout) :: model
        integer, intent(in) :: pdg
        real(default), intent(in) :: width
        type(particle_data_t), pointer :: prt
        prt => model_get_particle_ptr (model, pdg)
        if (associated (prt)) call particle_data_set_width (prt, width)
    end subroutine model_set_particle_width

```

Return the PDG code that matches a particle name.

```

<Models: public>+≡
    public :: model_get_particle_pdg

<Models: procedures>+≡
    function model_get_particle_pdg (model, name) result (pdg)
        integer :: pdg
        type(model_t), intent(in), target :: model
        type(string_t), intent(in) :: name
        integer :: i
        pdg = UNDEFINED
        do i = 1, size (model%prt)
            if (model%prt(i)%longname == name) then
                pdg = particle_data_get_pdg (model%prt(i)); exit
            else if (any (model%prt(i)%name == name)) then
                pdg = particle_data_get_pdg (model%prt(i)); exit
            else if (any (model%prt(i)%anti == name)) then
                pdg = - particle_data_get_pdg (model%prt(i)); exit
            end if
        end do
        if (pdg == UNDEFINED) then
            write (msg_buffer, "(1x,A,1x,A)") "Particle name =", char (name)
            call msg_message
            call msg_fatal (" Model '" // char (model%name) // "' // &
                " has no particle with this name")
        end if
    end function model_get_particle_pdg

```

Return a pointer to the variable list.

```

<Models: public>+≡

```

```

    public :: model_get_var_list_ptr
<Models: procedures>+≡
    function model_get_var_list_ptr (model) result (var_list)
        type(var_list_t), pointer :: var_list
        type(model_t), intent(in), target :: model
        var_list => model%var_list
    end function model_get_var_list_ptr

```

Vertex definition.

```

<Models: interfaces>+≡
    interface model_set_vertex
        module procedure model_set_vertex_pdg
        module procedure model_set_vertex_names
    end interface

<Models: procedures>+≡
    subroutine model_set_vertex_pdg (model, i, pdg)
        type(model_t), intent(inout), target :: model
        integer, intent(in) :: i
        integer, dimension(:), intent(in) :: pdg
        call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_pdg

    subroutine model_set_vertex_names (model, i, name)
        type(model_t), intent(inout), target :: model
        integer, intent(in) :: i
        type(string_t), dimension(:), intent(in) :: name
        integer, dimension(size(name)) :: pdg
        integer :: j
        do j = 1, size (name)
            pdg(j) = model_get_particle_pdg (model, name(j))
        end do
        call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_names

```

Lookup functions

```

<Models: public>+≡
    public :: model_match_vertex

<Models: procedures>+≡
    subroutine model_match_vertex (model, pdg1, pdg2, pdg3)
        type(model_t), intent(in) :: model
        integer, intent(in) :: pdg1, pdg2
        integer, dimension(:), allocatable, intent(out) :: pdg3
        call vertex_table_match (model%vt, pdg1, pdg2, pdg3)
    end subroutine model_match_vertex

<Models: public>+≡
    public :: model_check_vertex

<Models: procedures>+≡
    function model_check_vertex (model, pdg1, pdg2, pdg3) result (flag)
        logical :: flag

```

```

type(model_t), intent(in) :: model
integer, intent(in) :: pdg1, pdg2, pdg3
flag = vertex_table_check (model%vt, pdg1, pdg2, pdg3)
end function model_check_vertex

```

### 6.1.8 Reading models from file

This procedure defines the model-file syntax for the parser, returning an internal file (ifile).

Note that arithmetic operators are defined as keywords in the expression syntax, so we exclude them here.

*(Models: procedures)*+≡

```

subroutine define_model_file_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ model_def = model_name_def " // &
    "parameters derived_pars external_pars particles vertices")
  call ifile_append (ifile, "SEQ model_name_def = model model_name")
  call ifile_append (ifile, "KEY model")
  call ifile_append (ifile, "QUO model_name = '""'...'""')
  call ifile_append (ifile, "SEQ parameters = parameter_def*")
  call ifile_append (ifile, "SEQ parameter_def = parameter par_name " // &
    "'=' any_real_value")
  call ifile_append (ifile, "ALT any_real_value = " &
    // "neg_real_value | pos_real_value | real_value")
  call ifile_append (ifile, "SEQ neg_real_value = '-' real_value")
  call ifile_append (ifile, "SEQ pos_real_value = '+' real_value")
  call ifile_append (ifile, "KEY parameter")
  call ifile_append (ifile, "IDE par_name")
!   call ifile_append (ifile, "KEY '='")
!   call ifile_append (ifile, "REA par_value")
  call ifile_append (ifile, "SEQ derived_pars = derived_def*")
  call ifile_append (ifile, "SEQ derived_def = derived par_name " // &
    "'=' expr")
  call ifile_append (ifile, "KEY derived")
  call ifile_append (ifile, "SEQ external_pars = external_def*")
  call ifile_append (ifile, "SEQ external_def = external par_name")
  call ifile_append (ifile, "KEY external")
  call ifile_append (ifile, "SEQ particles = particle_def*")
  call ifile_append (ifile, "SEQ particle_def = particle prt_longname " // &
    "prt_pdg prt_details")
  call ifile_append (ifile, "KEY particle")
  call ifile_append (ifile, "IDE prt_longname")
  call ifile_append (ifile, "INT prt_pdg")
  call ifile_append (ifile, "ALT prt_details = prt_src | prt_properties")
  call ifile_append (ifile, "SEQ prt_src = like prt_longname prt_properties")
  call ifile_append (ifile, "KEY like")
  call ifile_append (ifile, "SEQ prt_properties = prt_property*")
  call ifile_append (ifile, "ALT prt_property = " // &
    "parton | invisible | gauge | left | right | " // &
    "prt_name | prt_anti | prt_tex_name | prt_tex_anti | " // &
    "prt_spin | prt_isospin | prt_charge | " // &
    "prt_color | prt_mass | prt_width")

```

```

call ifile_append (ifile, "KEY parton")
call ifile_append (ifile, "KEY invisible")
call ifile_append (ifile, "KEY gauge")
call ifile_append (ifile, "KEY left")
call ifile_append (ifile, "KEY right")
call ifile_append (ifile, "SEQ prt_name = name name_def+")
call ifile_append (ifile, "SEQ prt_anti = anti name_def+")
call ifile_append (ifile, "SEQ prt_tex_name = tex_name name_def")
call ifile_append (ifile, "SEQ prt_tex_anti = tex_anti name_def")
call ifile_append (ifile, "KEY name")
call ifile_append (ifile, "KEY anti")
call ifile_append (ifile, "KEY tex_name")
call ifile_append (ifile, "KEY tex_anti")
call ifile_append (ifile, "ALT name_def = name_string | name_id")
call ifile_append (ifile, "QUO name_string = '""'...'""')
call ifile_append (ifile, "IDE name_id")
call ifile_append (ifile, "SEQ prt_spin = spin frac")
call ifile_append (ifile, "KEY spin")
!   call ifile_append (ifile, "SEQ frac = signed_int div?")
!   call ifile_append (ifile, "ALT signed_int = " &
!       // "neg_int | pos_int | integer_literal")
!   call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
!   call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
!   call ifile_append (ifile, "KEY '-'")
!   call ifile_append (ifile, "KEY '+'")
!   call ifile_append (ifile, "INT int")
!   call ifile_append (ifile, "SEQ div = '/' integer_literal")
!   call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "SEQ prt_isospin = isospin frac")
call ifile_append (ifile, "KEY isospin")
call ifile_append (ifile, "SEQ prt_charge = charge frac")
call ifile_append (ifile, "KEY charge")
call ifile_append (ifile, "SEQ prt_color = color integer_literal")
call ifile_append (ifile, "KEY color")
call ifile_append (ifile, "SEQ prt_mass = mass par_name")
call ifile_append (ifile, "KEY mass")
call ifile_append (ifile, "SEQ prt_width = width par_name")
call ifile_append (ifile, "KEY width")
call ifile_append (ifile, "SEQ vertices = vertex_def*")
call ifile_append (ifile, "SEQ vertex_def = vertex name_def+")
call ifile_append (ifile, "KEY vertex")
call define_expr_syntax (ifile, particles=.false., analysis=.false.)
end subroutine define_model_file_syntax

```

The model-file syntax and lexer are fixed, therefore stored as module variables:

```

<Models: variables>≡
    type(syntax_t), target, save :: syntax_model_file

<Models: public>+≡
    public :: syntax_model_file_init

<Models: procedures>+≡
    subroutine syntax_model_file_init ()
        type(ifile_t) :: ifile

```

```

        call define_model_file_syntax (ifile)
        call syntax_init (syntax_model_file, ifile)
        call ifile_final (ifile)
    end subroutine syntax_model_file_init

<Models: procedures>+≡
    subroutine lexer_init_model_file (lexer)
        type(lexer_t), intent(out) :: lexer
        call lexer_init (lexer, &
            comment_chars = "#!", &
            quote_chars = '"{', &
            quote_match = '"}', &
            single_chars = ":()", &
            special_class = (/ "+-*/^", "<=> " /) , &
            keyword_list = syntax_get_keyword_list_ptr (syntax_model_file))
    end subroutine lexer_init_model_file

<Models: public>+≡
    public :: syntax_model_file_final

<Models: procedures>+≡
    subroutine syntax_model_file_final ()
        call syntax_final (syntax_model_file)
    end subroutine syntax_model_file_final

<Models: public>+≡
    public :: syntax_model_file_write

<Models: procedures>+≡
    subroutine syntax_model_file_write (unit)
        integer, intent(in), optional :: unit
        call syntax_write (syntax_model_file, unit)
    end subroutine syntax_model_file_write

<Models: public>+≡
    public :: model_read

<Models: procedures>+≡
    subroutine model_read (model, filename, os_data, exist)
        type(model_t), intent(out), target :: model
        type(string_t), intent(in) :: filename
        type(os_data_t), intent(in) :: os_data
        logical, intent(out) :: exist
        type(string_t) :: file
        type(stream_t), target :: stream
        type(lexer_t) :: lexer
        integer :: unit
        character(32) :: model_md5sum
        type(parse_tree_t) :: parse_tree
        type(parse_node_t), pointer :: nd_model_def, nd_model_name_def
        type(parse_node_t), pointer :: nd_model_arg
        type(parse_node_t), pointer :: nd_parameters, nd_derived_pars
        type(parse_node_t), pointer :: nd_external_pars
        type(parse_node_t), pointer :: nd_particles, nd_vertices

```

```

type(string_t) :: model_name, lib_name
integer :: n_par, n_der, n_ext, n_prt, n_vtx
real(c_default_float), dimension(:), allocatable :: par
integer :: i
type(parse_node_t), pointer :: nd_par_def
type(parse_node_t), pointer :: nd_der_def
type(parse_node_t), pointer :: nd_ext_def
type(parse_node_t), pointer :: nd_prt
type(parse_node_t), pointer :: nd_vtx
type(pdg_array_t) :: prt_undefined
file = filename
inquire (file=char(file), exist=exist)
if ((.not. exist) .and. (.not. os_data%use_testfiles)) then
    file = os_data%whizard_modelpath_local // "/" // filename
    inquire (file = char (file), exist = exist)
end if
if (.not. exist) then
    file = os_data%whizard_modelpath // "/" // filename
    inquire (file = char (file), exist = exist)
end if
if (.not. exist) then
    call msg_fatal ("Model file '" // char (filename) // "' not found")
    return
end if
if (logging) call msg_message ("Reading model file '" // char (file) // "'")
call lexer_init_model_file (lexer)
unit = free_unit ()
open (file=char(file), unit=unit, action="read", status="old")
model_md5sum = md5sum (unit)
close (unit)
call stream_init (stream, char (file))
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax_model_file, lexer)
call stream_final (stream)
call lexer_final (lexer)
!    call parse_tree_write (parse_tree)
nd_model_def => parse_tree_get_root_ptr (parse_tree)
nd_model_name_def => parse_node_get_sub_ptr (nd_model_def)
model_name = parse_node_get_string &
    (parse_node_get_sub_ptr (nd_model_name_def, 2))
nd_parameters => parse_node_get_next_ptr (nd_model_name_def)
if (associated (nd_parameters)) then
    if (parse_node_get_rule_key (nd_parameters) == "parameters") then
        n_par = parse_node_get_n_sub (nd_parameters)
        nd_par_def => parse_node_get_sub_ptr (nd_parameters)
        nd_derived_pars => parse_node_get_next_ptr (nd_parameters)
    else
        n_par = 0
        nd_derived_pars => nd_parameters
        nd_parameters => null ()
    end if
else
    n_par = 0
    nd_derived_pars => null ()

```



```

end if
if (associated (nd_derived_pars)) then
  if (parse_node_get_rule_key (nd_derived_pars) == "derived_pars") then
    n_der = parse_node_get_n_sub (nd_derived_pars)
    nd_der_def => parse_node_get_sub_ptr (nd_derived_pars)
    nd_external_pars => parse_node_get_next_ptr (nd_derived_pars)
  else
    n_der = 0
    nd_external_pars => nd_derived_pars
    nd_derived_pars => null ()
  end if
else
  n_der = 0
  nd_external_pars => null ()
end if
if (associated (nd_external_pars)) then
  if (parse_node_get_rule_key (nd_external_pars) == "external_pars") then
    n_ext = parse_node_get_n_sub (nd_external_pars)
    lib_name = "external." // model_name
    nd_ext_def => parse_node_get_sub_ptr (nd_external_pars)
    nd_particles => parse_node_get_next_ptr (nd_external_pars)
  else
    n_ext = 0
    lib_name = ""
    nd_particles => nd_external_pars
    nd_external_pars => null ()
  end if
else
  n_ext = 0
  lib_name = ""
  nd_particles => null ()
end if
if (associated (nd_particles)) then
  if (parse_node_get_rule_key (nd_particles) == "particles") then
    n_prt = parse_node_get_n_sub (nd_particles)
    nd_prt => parse_node_get_sub_ptr (nd_particles)
    nd_vertices => parse_node_get_next_ptr (nd_particles)
  else
    n_prt = 0
    nd_vertices => nd_particles
    nd_particles => null ()
  end if
else
  n_prt = 0
  nd_vertices => null ()
end if
if (associated (nd_vertices)) then
  n_vtx = parse_node_get_n_sub (nd_vertices)
  nd_vtx => parse_node_get_sub_ptr (nd_vertices)
else
  n_vtx = 0
end if
call model_init (model, model_name, lib_name, os_data, &
  n_par + n_der + n_ext, n_prt, n_vtx)

```

```

model%md5sum = model_md5sum
do i = 1, n_par
    call model_read_parameter (model, i, nd_par_def)
    nd_par_def => parse_node_get_next_ptr (nd_par_def)
end do
do i = n_par + 1, n_par + n_der
    call model_read_derived (model, i, nd_der_def)
    nd_der_def => parse_node_get_next_ptr (nd_der_def)
end do
do i = n_par + n_der + 1, n_par + n_der + n_ext
    call model_read_external (model, i, nd_ext_def)
    nd_ext_def => parse_node_get_next_ptr (nd_ext_def)
end do
if (associated (model% init_external_parameters)) then
    call model_parameters_to_c_array (model, par)
    call model% init_external_parameters (par)
    call model_parameters_from_c_array (model, par)
end if
prt_undefined = UNDEFINED
call var_list_append_pdg_array &
    (model%var_list, var_str ("particle"), &
    prt_undefined, locked = .true., intrinsic=.true.)
do i = 1, n_prt
    call model_read_particle (model, i, nd_prt)
    nd_prt => parse_node_get_next_ptr (nd_prt)
end do
do i = 1, n_vtx
    call model_read_vertex (model, i, nd_vtx)
    nd_vtx => parse_node_get_next_ptr (nd_vtx)
end do
call parse_tree_final (parse_tree)
call var_list_append_pdg_array &
    (model%var_list, var_str ("charged"), &
    particle_data_get_charged_pdg (model%prt), locked = .true., &
    intrinsic=.true.)
call var_list_append_pdg_array &
    (model%var_list, var_str ("colored"), &
    particle_data_get_colored_pdg (model%prt), locked = .true., &
    intrinsic=.true.)
end subroutine model_read

```

Parameters are real values (literal) with an optional unit.

*(Models: procedures)*+≡

```

subroutine model_read_parameter (model, i, node)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in), target :: node
    type(parse_node_t), pointer :: node_name, node_val
    type(string_t) :: name
    node_name => parse_node_get_sub_ptr (node, 2)
    name = parse_node_get_string (node_name)
    node_val => parse_node_get_next_ptr (node_name, 2)
    call model_set_parameter (model, i, name, node_val, constant=.true.)

```

```
end subroutine model_read_parameter
```

Derived parameters have any numeric expression as their definition.

*(Models: procedures)*+≡

```
subroutine model_read_derived (model, i, node)
  type(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in), target :: node
  type(string_t) :: name
  type(parse_node_t), pointer :: pn_expr
  name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  pn_expr => parse_node_get_sub_ptr (node, 4)
  call model_set_parameter (model, i, name, pn_expr, constant=.false.)
end subroutine model_read_derived
```

External parameters have no definition; they are handled by an external library.

*(Models: procedures)*+≡

```
subroutine model_read_external (model, i, node)
  type(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in), target :: node
  type(string_t) :: name
  name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  call model_set_parameter_external (model, i, name)
end subroutine model_read_external
```

*(Models: procedures)*+≡

```
subroutine model_read_particle (model, i, node)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in) :: node
  type(parse_node_t), pointer :: nd_src, nd_props, nd_prop
  type(string_t) :: longname
  integer :: pdg
  type(string_t) :: name_src
  type(string_t), dimension(:), allocatable :: name
  longname = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  pdg = parse_node_get_integer (parse_node_get_sub_ptr (node, 3))
  call model_init_particle (model, i, longname, pdg)
  nd_src => parse_node_get_sub_ptr (node, 4)
  if (associated (nd_src)) then
    if (parse_node_get_rule_key (nd_src) == "prt_src") then
      name_src = parse_node_get_string (parse_node_get_sub_ptr (nd_src, 2))
      call model_copy_particle_data (model, i, name_src)
      nd_props => parse_node_get_sub_ptr (nd_src, 3)
    else
      nd_props => nd_src
    end if
  end if
  nd_prop => parse_node_get_sub_ptr (nd_props)
  do while (associated (nd_prop))
    select case (char (parse_node_get_rule_key (nd_prop)))
    case ("invisible")
      call model_set_particle_data (model, i, is_visible=.false.)
    end select
  end do
```

```

case ("parton")
    call model_set_particle_data (model, i, is_parton=.true.)
case ("gauge")
    call model_set_particle_data (model, i, is_gauge=.true.)
case ("left")
    call model_set_particle_data (model, i, is_left_handed=.true.)
case ("right")
    call model_set_particle_data (model, i, is_right_handed=.true.)
case ("prt_name")
    call read_names (nd_prop, name)
    call model_set_particle_data (model, i, name=name)
case ("prt_anti")
    call read_names (nd_prop, name)
    call model_set_particle_data (model, i, anti=name)
case ("prt_tex_name")
    call model_set_particle_data (model, i, &
        tex_name = parse_node_get_string &
        (parse_node_get_sub_ptr (nd_prop, 2)))
case ("prt_tex_anti")
    call model_set_particle_data (model, i, &
        tex_anti = parse_node_get_string &
        (parse_node_get_sub_ptr (nd_prop, 2)))
case ("prt_spin")
    call model_set_particle_data (model, i, &
        spin_type = read_frac &
        (parse_node_get_sub_ptr (nd_prop, 2), 2))
case ("prt_isospin")
    call model_set_particle_data (model, i, &
        isospin_type = read_frac &
        (parse_node_get_sub_ptr (nd_prop, 2), 2))
case ("prt_charge")
    call model_set_particle_data (model, i, &
        charge_type = read_frac &
        (parse_node_get_sub_ptr (nd_prop, 2), 3))
case ("prt_color")
    call model_set_particle_data (model, i, &
        color_type = parse_node_get_integer &
        (parse_node_get_sub_ptr (nd_prop, 2)))
case ("prt_mass")
    call model_set_particle_data (model, i, &
        mass_src = model_get_parameter_ptr &
        (model, parse_node_get_string &
        (parse_node_get_sub_ptr (nd_prop, 2))))
case ("prt_width")
    call model_set_particle_data (model, i, &
        width_src = model_get_parameter_ptr &
        (model, parse_node_get_string &
        (parse_node_get_sub_ptr (nd_prop, 2))))
case default
    call msg_bug (" Unknown particle property '" &
        // char (parse_node_get_rule_key (nd_prop)) // "'")
end select
if (allocated (name)) deallocate (name)
nd_prop => parse_node_get_next_ptr (nd_prop)

```

```

        end do
    end if
    call model_freeze_particle_data (model, i)
end subroutine model_read_particle

```

*<Models: procedures>+≡*

```

subroutine model_read_vertex (model, i, node)
    type(model_t), intent(inout) :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in) :: node
    type(string_t), dimension(:), allocatable :: name
    call read_names (node, name)
    call model_set_vertex (model, i, name)
end subroutine model_read_vertex

```

*<Models: procedures>+≡*

```

subroutine read_names (node, name)
    type(parse_node_t), intent(in) :: node
    type(string_t), dimension(:), allocatable, intent(inout) :: name
    type(parse_node_t), pointer :: nd_name
    integer :: n_names, i
    n_names = parse_node_get_n_sub (node) - 1
    allocate (name (n_names))
    nd_name => parse_node_get_sub_ptr (node, 2)
    do i = 1, n_names
        name(i) = parse_node_get_string (nd_name)
        nd_name => parse_node_get_next_ptr (nd_name)
    end do
end subroutine read_names

```

*<Models: procedures>+≡*

```

function read_frac (nd_frac, base) result (qn_type)
    integer :: qn_type
    type(parse_node_t), intent(in) :: nd_frac
    integer, intent(in) :: base
    type(parse_node_t), pointer :: nd_num, nd_den
    integer :: num, den
    nd_num => parse_node_get_sub_ptr (nd_frac)
    nd_den => parse_node_get_next_ptr (nd_num)
    select case (char (parse_node_get_rule_key (nd_num)))
    case ("integer_literal")
        num = parse_node_get_integer (nd_num)
    case ("neg_int")
        num = - parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
    case ("pos_int")
        num = parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
    case default
        call parse_tree_bug (nd_num, "int|neg_int|pos_int")
    end select
    if (associated (nd_den)) then
        den = parse_node_get_integer (parse_node_get_sub_ptr (nd_den, 2))
    else
        den = 1
    end if
end function read_frac

```

```

end if
if (den == 1) then
  qn_type = sign (1 + abs (num) * base, num)
else if (den == base) then
  qn_type = sign (abs (num) + 1, num)
else
  call parse_node_write_rec (nd_frac)
  call msg_fatal (" Fractional quantum number: wrong denominator")
end if
end function read_frac

```

### 6.1.9 Model list

List of currently active models

```

<Models: types>+≡
type :: model_entry_t
  type(model_t) :: model
  type(model_entry_t), pointer :: next => null ()
end type model_entry_t

```

```

<Models: types>+≡
type :: model_list_t
  type(model_entry_t), pointer :: first => null ()
  type(model_entry_t), pointer :: last => null ()
end type model_list_t

```

The model list is stored as a module variable. Thus, the operations acting on the list do not have the model list as an argument.

```

<Models: variables>+≡
type(model_list_t), target, save :: model_list

```

Write an account of the model list.

```

<Models: public>+≡
public :: model_list_write

<Models: procedures>+≡
subroutine model_list_write (unit, verbose)
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  type(model_entry_t), pointer :: current
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, *) "List of models:"
  current => model_list%first
  if (associated (current)) then
    do while (associated (current))
      write (u, *)
      call model_write (current%model, unit, verbose)
      current => current%next
    end do
  else

```

```

        write (u, *) " [empty]"
    end if
end subroutine model_list_write

```

Add a new model with given name to the list, if it does not yet exist. If successful, return a pointer to the new model.

*(Models: procedures)*+≡

```

subroutine model_list_add (name, os_data, n_par, n_prt, n_vtx, model)
    type(string_t), intent(in) :: name
    type(os_data_t), intent(in) :: os_data
    integer, intent(in) :: n_par, n_prt, n_vtx
    type(model_t), pointer :: model
    type(model_entry_t), pointer :: current
    if (model_list_model_exists (name)) then
        model => null ()
    else
        allocate (current)
        if (associated (model_list%first)) then
            model_list%last%next => current
        else
            model_list%first => current
        end if
        model_list%last => current
        model => current%model
        call model_init (model, name, var_str (""), os_data, &
            n_par, n_prt, n_vtx)
    end if
end subroutine model_list_add

```

Read a new model from file and add to the list, if it does not yet exist. Finalize the model by allocating the vertex table. Return a pointer to the new model. If unsuccessful, return the original pointer.

*(Models: public)*+≡

```

public :: model_list_read_model

```

*(Models: procedures)*+≡

```

subroutine model_list_read_model (name, filename, os_data, model)
    type(string_t), intent(in) :: name, filename
    type(os_data_t), intent(in) :: os_data
    type(model_t), pointer :: model
    type(model_entry_t), pointer :: current
    logical :: exist
    if (.not. model_list_model_exists (name)) then
        allocate (current)
        call model_read (current%model, filename, os_data, exist)
        if (.not. exist) return
        if (current%model%name /= name) then
            call msg_fatal ("Model file '" // char (filename) // &
                "' contains model '" // char (current%model%name) // &
                "' instead of '" // char (name) // "'")
            call model_final (current%model); deallocate (current)
            return
        end if
    end if

```

```

        if (associated (model_list%first)) then
            model_list%last%next => current
        else
            model_list%first => current
        end if
        model_list%last => current
        call vertex_table_init &
            (current%model%vt, current%model%prt, current%model%vtx)
        model => current%model
    else
        model => model_list_get_model_ptr (name)
    end if
end subroutine model_list_read_model

```

Check if a model exists by examining the list

```

<Models: public>+≡
    public :: model_list_model_exists

<Models: procedures>+≡
    function model_list_model_exists (name) result (exists)
        logical :: exists
        type(string_t), intent(in) :: name
        type(model_entry_t), pointer :: current
        current => model_list%first
        do while (associated (current))
            if (current%model%name == name) then
                exists = .true.
                return
            end if
            current => current%next
        end do
        exists = .false.
    end function model_list_model_exists

```

Return a pointer to a named model

```

<Models: public>+≡
    public :: model_list_get_model_ptr

<Models: procedures>+≡
    function model_list_get_model_ptr (name) result (model)
        type(model_t), pointer :: model
        type(string_t), intent(in) :: name
        type(model_entry_t), pointer :: current
        current => model_list%first
        do while (associated (current))
            if (current%model%name == name) then
                model => current%model
                return
            end if
            current => current%next
        end do
        model => null ()
    end function model_list_get_model_ptr

```



Delete the list of models

```
<Models: public>+≡
    public :: model_list_final

<Models: procedures>+≡
    subroutine model_list_final ()
        type(model_entry_t), pointer :: current
        model_list%last => null ()
        do while (associated (model_list%first))
            current => model_list%first
            model_list%first => model_list%first%next
            call model_final (current%model)
            deallocate (current)
        end do
    end subroutine model_list_final
```

### 6.1.10 Test

```
<Models: public>+≡
    public :: models_test

<Models: procedures>+≡
    subroutine models_test ()
        type(os_data_t), pointer :: os_data => null ()
        call syntax_model_file_init ()
        call syntax_write (syntax_model_file)
        print *
        allocate (os_data)
        call os_data_init (os_data)
        call models_test1 (os_data)
        call models_test2 (os_data)
        !!! Try to catch the gfortran 4.5.0+(?) seg fault
        !!! call model_list_write (verbose=.true.)
        call model_list_write ()
        call model_list_final ()
        call syntax_model_file_final ()
        deallocate (os_data)
    end subroutine models_test

    subroutine models_test1 (os_data)
        type(os_data_t), intent(in) :: os_data
        type(model_t), pointer :: model
        type(string_t) :: model_name
        type(string_t) :: x_longname
        type(string_t), dimension(2) :: parname
        type(string_t), dimension(2) :: x_name
        type(string_t), dimension(1) :: x_anti
        type(string_t) :: x_tex_name, x_tex_anti
        type(string_t) :: y_longname
        type(string_t), dimension(2) :: y_name
        type(string_t) :: y_tex_name
        model_name = "Test model"
        call model_list_add (model_name, os_data, 2, 2, 3, model)
        parname(1) = "mx"
```

```

parname(2) = "coup"
call model_set_parameter (model, 1, parname(1), 10._default)
call model_set_parameter (model, 2, parname(2), 1.3_default)
print *
x_longname = "X_LEPTON"
x_name(1) = "X"
x_name(2) = "x"
x_anti(1) = "Xbar"
x_tex_name = "X^+"
x_tex_anti = "X^-"
call model_init_particle (model, 1, x_longname, 99)
call model_set_particle_data (model, 1, &
    .true., .false., .false., .false., .false., &
    x_name, x_anti, x_tex_name, x_tex_anti, &
    SPINOR, -3, 2, 1, model_get_parameter_ptr (model, parname(1)))
y_longname = "Y_COLORON"
y_name(1) = "Y"
y_name(2) = "yc"
y_tex_name = "Y^0"
call model_init_particle (model, 2, y_longname, 97)
call model_set_particle_data (model, 2, &
    .false., .false., .true., .false., .false., &
    name=y_name, tex_name=y_tex_name, &
    spin_type=SCALAR, isospin_type=2, charge_type=1, color_type=8)
call model_set_vertex (model, 1, (/ 99, 99, 99 /))
call model_set_vertex (model, 2, (/ 99, 99, 99, 99 /))
call model_set_vertex (model, 3, (/ 99, 97, 99 /))
end subroutine models_test1

subroutine models_test2 (os_data)
    type(os_data_t), intent(in) :: os_data
    type(string_t) :: name, filename
    type(model_t), pointer :: model
    name = "SM"
    filename = "SM.mdl"
    call model_list_read_model (name, filename, os_data, model)
end subroutine models_test2

```

## Chapter 7

# Quantum Numbers

We introduce separate types and modules for particle quantum numbers and use them for defining independent particles and entangled states.

**helicities** Types and methods for spin density matrices.

**colors** Dealing with colored particles, using the color-flow representation.

**flavors** PDG codes and particle properties, depends on the model.

**quantum\_states** Quantum numbers and density matrices for entangled particle systems.

## 7.1 Helicities

This module defines types and tools for dealing with helicity information.

```
<helicities.f90>≡  
  <File header>  
  
  module helicities  
  
    <Use file utils>  
  
    <Standard module head>  
  
    <Helicities: public>  
  
    <Helicities: types>  
  
    <Helicities: interfaces>  
  
    contains  
  
    <Helicities: procedures>  
  
  end module helicities
```

### 7.1.1 Helicity types

Helicities may be defined or undefined, corresponding to a polarized or unpolarized state. Each helicity is actually a pair of helicities, corresponding to an entry in the spin density matrix. Obviously, diagonal entries are distinguished. In addition, we have a ghost flag that would apply to FP ghosts in particular.

```
<Helicities: public>≡  
  public :: helicity_t  
  
<Helicities: types>≡  
  type :: helicity_t  
    private  
    logical :: defined = .false.  
    integer :: h1, h2  
    logical :: ghost = .false.  
  end type helicity_t
```

Initializers:

```
<Helicities: public>+≡  
  public :: helicity_init  
  
<Helicities: interfaces>≡  
  interface helicity_init  
    module procedure helicity_init0, helicity_init0g  
    module procedure helicity_init1, helicity_init1g  
    module procedure helicity_init2, helicity_init2g  
  end interface
```

```

<Helicities: procedures>+≡
  elemental subroutine helicity_init0 (hel)
    type(helicity_t), intent(out) :: hel
  end subroutine helicity_init0

  elemental subroutine helicity_init0g (hel, ghost)
    type(helicity_t), intent(out) :: hel
    logical, intent(in) :: ghost
    hel%ghost = ghost
  end subroutine helicity_init0g

  elemental subroutine helicity_init1 (hel, h)
    type(helicity_t), intent(out) :: hel
    integer, intent(in) :: h
    hel%defined = .true.
    hel%h1 = h
    hel%h2 = h
  end subroutine helicity_init1

  elemental subroutine helicity_init1g (hel, h, ghost)
    type(helicity_t), intent(out) :: hel
    integer, intent(in) :: h
    logical, intent(in) :: ghost
    call helicity_init1 (hel, h)
    hel%ghost = ghost
  end subroutine helicity_init1g

  elemental subroutine helicity_init2 (hel, h2, h1)
    type(helicity_t), intent(out) :: hel
    integer, intent(in) :: h1, h2
    hel%defined = .true.
    hel%h2 = h2
    hel%h1 = h1
  end subroutine helicity_init2

  elemental subroutine helicity_init2g (hel, h2, h1, ghost)
    type(helicity_t), intent(out) :: hel
    integer, intent(in) :: h1, h2
    logical, intent(in) :: ghost
    call helicity_init2 (hel, h2, h1)
    hel%ghost = ghost
  end subroutine helicity_init2g

```

Set the ghost property separately:

```

<Helicities: public>+≡
  public :: helicity_set_ghost

<Helicities: procedures>+≡
  elemental subroutine helicity_set_ghost (hel, ghost)
    type(helicity_t), intent(inout) :: hel
    logical, intent(in) :: ghost
    hel%ghost = ghost
  end subroutine helicity_set_ghost

```

Undefine:

```
<Helicities: public>+≡
    public :: helicity_undefine

<Helicities: procedures>+≡
    elemental subroutine helicity_undefine (hel)
        type(helicity_t), intent(inout) :: hel
        hel%defined = .false.
        hel%ghost = .false.
    end subroutine helicity_undefine
```

Diagonalize by removing the second entry (use with care!)

```
<Helicities: public>+≡
    public :: helicity_diagonalize

<Helicities: procedures>+≡
    elemental subroutine helicity_diagonalize (hel)
        type(helicity_t), intent(inout) :: hel
        hel%h2 = hel%h1
    end subroutine helicity_diagonalize
```

Output (no linebreak). No output if undefined.

```
<Helicities: public>+≡
    public :: helicity_write

<Helicities: procedures>+≡
    subroutine helicity_write (hel, unit)
        type(helicity_t), intent(in) :: hel
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        if (hel%defined) then
            if (hel%ghost) then
                write (u, "(A)", advance="no") "h*"
            else
                write (u, "(A)", advance="no") "h("
            end if
            write (u, "(IO)", advance="no") hel%h1
            if (hel%h1 /= hel%h2) then
                write (u, "(A)", advance="no") "|"
                write (u, "(IO)", advance="no") hel%h2
            end if
            write (u, "(A)", advance="no") ")"
        else if (hel%ghost) then
            write (u, "(A)", advance="no") "h*"
        end if
    end subroutine helicity_write
```

Binary I/O. Write contents only if defined.

```
<Helicities: public>+≡
    public :: helicity_write_raw
    public :: helicity_read_raw
```

```

<Helicities: procedures>+≡
  subroutine helicity_write_raw (hel, u)
    type(helicity_t), intent(in) :: hel
    integer, intent(in) :: u
    write (u) hel%defined
    if (hel%defined) then
      write (u) hel%h1, hel%h2
      write (u) hel%ghost
    end if
  end subroutine helicity_write_raw

  subroutine helicity_read_raw (hel, u, iostat)
    type(helicity_t), intent(out) :: hel
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    read (u, iostat=iostat) hel%defined
    if (hel%defined) then
      read (u, iostat=iostat) hel%h1, hel%h2
      read (u, iostat=iostat) hel%ghost
    end if
  end subroutine helicity_read_raw

```

### 7.1.2 Predicates

Check if the helicity is defined:

```

<Helicities: public>+≡
  public :: helicity_is_defined

<Helicities: procedures>+≡
  elemental function helicity_is_defined (hel) result (defined)
    logical :: defined
    type(helicity_t), intent(in) :: hel
    defined = hel%defined
  end function helicity_is_defined

```

Return true if the two helicities are equal or the particle is unpolarized:

```

<Helicities: public>+≡
  public :: helicity_is_diagonal

<Helicities: procedures>+≡
  elemental function helicity_is_diagonal (hel) result (diagonal)
    logical :: diagonal
    type(helicity_t), intent(in) :: hel
    if (hel%defined) then
      diagonal = hel%h1 == hel%h2
    else
      diagonal = .true.
    end if
  end function helicity_is_diagonal

```

Return the ghost flag

```

<Helicities: public>+≡
  public :: helicity_is_ghost

```

```

<Helicities: procedures>+≡
  elemental function helicity_is_ghost (hel) result (ghost)
    logical :: ghost
    type(helicity_t), intent(in) :: hel
    ghost = hel%ghost
  end function helicity_is_ghost

```

### 7.1.3 Accessing contents

This returns a two-element array and thus cannot be elemental. The result is unpredictable if the helicity is undefined.

```

<Helicities: public>+≡
  public :: helicity_get

<Helicities: procedures>+≡
  pure function helicity_get (hel) result (h)
    integer, dimension(2) :: h
    type(helicity_t), intent(in) :: hel
    h(1) = hel%h2
    h(2) = hel%h1
  end function helicity_get

```

### 7.1.4 Comparisons

When comparing helicities, if either one is undefined, they are considered to match. In other words, an unpolarized particle matches any polarization. In the `dmatch` variant, it matches only diagonal helicity.

The ghost flag is ignored when matching, but matters when testing for equality.

```

<Helicities: public>+≡
  public :: operator(.match.)
  public :: operator(.dmatch.)
  public :: operator(==)
  public :: operator(/=)

<Helicities: interfaces>+≡
  interface operator(.match.)
    module procedure helicity_match
  end interface
  interface operator(.dmatch.)
    module procedure helicity_match_diagonal
  end interface
  interface operator(==)
    module procedure helicity_eq
  end interface
  interface operator(/=)
    module procedure helicity_neq
  end interface

```



```

<Helicities: procedures>+=
  elemental function helicity_match (hel1, hel2) result (eq)
    logical :: eq
    type(helicity_t), intent(in) :: hel1, hel2
    if (hel1%defined .and. hel2%defined) then
      eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
    else
      eq = .true.
    end if
  end function helicity_match

  elemental function helicity_match_diagonal (hel1, hel2) result (eq)
    logical :: eq
    type(helicity_t), intent(in) :: hel1, hel2
    if (hel1%defined .and. hel2%defined) then
      eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
    else if (hel1%defined) then
      eq = hel1%h1 == hel1%h2
    else if (hel2%defined) then
      eq = hel2%h1 == hel2%h2
    else
      eq = .true.
    end if
  end function helicity_match_diagonal

<Helicities: procedures>+=
  elemental function helicity_eq (hel1, hel2) result (eq)
    logical :: eq
    type(helicity_t), intent(in) :: hel1, hel2
    if (hel1%defined .and. hel2%defined) then
      eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2) &
        .and. (hel1%ghost .eqv. hel2%ghost)
    else if (.not. hel1%defined .and. .not. hel2%defined) then
      eq = hel1%ghost .eqv. hel2%ghost
    else
      eq = .false.
    end if
  end function helicity_eq

<Helicities: procedures>+=
  elemental function helicity_neq (hel1, hel2) result (neq)
    logical :: neq
    type(helicity_t), intent(in) :: hel1, hel2
    if (hel1%defined .and. hel2%defined) then
      neq = (hel1%h1 /= hel2%h1) .or. (hel1%h2 /= hel2%h2) &
        .or. (hel1%ghost .neqv. hel2%ghost)
    else if (.not. hel1%defined .and. .not. hel2%defined) then
      neq = hel1%ghost .neqv. hel2%ghost
    else
      neq = .true.
    end if
  end function helicity_neq

```

### 7.1.5 Tools

Merge two helicity objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input helicities were defined and diagonal. The handling of ghost flags is not well-defined; one should verify beforehand that they match.

```
<Helicities: public>+≡
    public :: operator(.merge.)

<Helicities: interfaces>+≡
    interface operator(.merge.)
        module procedure merge_helicities
    end interface

<Helicities: procedures>+≡
    elemental function merge_helicities (hel1, hel2) result (hel)
        type(helicity_t) :: hel
        type(helicity_t), intent(in) :: hel1, hel2
        if (helicity_is_defined (hel1) .and. helicity_is_defined (hel2)) then
            call helicity_init2g (hel, hel2%h1, hel1%h1, hel1%ghost)
        else if (helicity_is_defined (hel1)) then
            hel = hel1
        else if (helicity_is_defined (hel2)) then
            hel = hel2
        end if
    end function merge_helicities
```

## 7.2 Colors

This module defines a type and tools for dealing with color information.

Each particle can have zero or more (in practice, usually not more than two) color indices. Color indices are positive; flow direction can be determined from the particle nature.

While parton shower matrix elements are diagonal in color, some special applications (e.g., subtractions for NLO matrix elements) require non-diagonal color matrices.

```
<colors.f90>≡  
  <File header>  
  
  module colors  
  
    <Use kinds>  
    <Use file utils>  
    use diagnostics !NODEP!  
  
    <Standard module head>  
  
    <Colors: public>  
  
    <Colors: types>  
  
    <Colors: interfaces>  
  
    contains  
  
    <Colors: procedures>  
  
  end module colors
```

### 7.2.1 The color type

A particle may have an arbitrary number of color indices (in practice, from zero to two, but more are possible). This object acts as a container.

The fact that color comes as an array prohibits elemental procedures in some places. (May add interfaces and multi versions where necessary.)

The color may be undefined; this corresponds to unallocated arrays.

NOTE: Due to a compiler bug in nagfor 5.2, we do not use allocatable but fixed-size arrays with dimension 2. Only nonzero entries count. This may be more efficient anyway, but gives up some flexibility. However, the squaring algorithm currently works only for singlets, (anti)triplets and octets anyway, so two components are enough.

```
<Colors: public>≡  
  public :: color_t  
  
<Colors: types>≡  
  type :: color_t  
    private  
    !   integer, dimension(:), allocatable :: c1, c2  
    integer, dimension(2) :: c1 = 0, c2 = 0
```

```

        logical :: ghost = .false.
    end type color_t

```

Initializers:

```

<Colors: public>+≡
    public :: color_init

<Colors: interfaces>≡
    interface color_init
        module procedure color_init_undefined, color_init_undefined_ghost
        module procedure color_init_array, color_init_array_ghost
        module procedure color_init_arrays, color_init_arrays_ghost
    end interface

```

Undefined color: array remains unallocated

```

<Colors: procedures>≡
    pure subroutine color_init_undefined (col)
        type(color_t), intent(out) :: col
    end subroutine color_init_undefined

    pure subroutine color_init_undefined_ghost (col, ghost)
        type(color_t), intent(out) :: col
        logical, intent(in) :: ghost
        col%ghost = ghost
    end subroutine color_init_undefined_ghost

```

This defines color from an arbitrary length color array, suitable for any representation. We may have two color arrays (non-diagonal matrix elements). This cannot be elemental. The third version assigns an array of colors, using a two-dimensional array as input.

```

<Colors: procedures>+≡
    pure subroutine color_init_array (col, c1)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1
    !    allocate (col%c1 (size (c1)))
    !    allocate (col%c2 (size (c1)))
        col%c1 = pack (c1, c1 /= 0, col%c1)
        col%c2 = col%c1
    end subroutine color_init_array

    pure subroutine color_init_array_ghost (col, c1, ghost)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1
        logical, intent(in) :: ghost
        call color_init_array (col, c1)
        col%ghost = ghost
    end subroutine color_init_array_ghost

    pure subroutine color_init_arrays (col, c1, c2)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1, c2
        if (size (c1) == size (c2)) then

```

```

!      allocate (col%c1 (size (c1)))
!      allocate (col%c2 (size (c2)))
      col%c1 = pack (c1, c1 /= 0, col%c1)
      col%c2 = pack (c2, c2 /= 0, col%c2)
    else if (size (c1) /= 0) then
!      allocate (col%c1 (size (c1)))
!      allocate (col%c2 (size (c1)))
      col%c1 = pack (c1, c1 /= 0, col%c1)
      col%c2 = col%c1
    else if (size (c2) /= 0) then
!      allocate (col%c1 (size (c2)))
!      allocate (col%c2 (size (c2)))
      col%c1 = pack (c2, c2 /= 0, col%c2)
      col%c2 = col%c1
    end if
  end subroutine color_init_arrays

  pure subroutine color_init_arrays_ghost (col, c1, c2, ghost)
    type(color_t), intent(out) :: col
    integer, dimension(:), intent(in) :: c1, c2
    logical, intent(in) :: ghost
    call color_init_arrays (col, c1, c2)
    col%ghost = ghost
  end subroutine color_init_arrays_ghost

```

This version is restricted to singlets, triplets, antitriplets, and octets: The input contains the color and anticolor index, each of the may be zero.

```

<Colors: public>+≡
  public :: color_init_col_acl

<Colors: procedures>+≡
  elemental subroutine color_init_col_acl (col, col_in, acl_in)
    type(color_t), intent(out) :: col
    integer, intent(in) :: col_in, acl_in
    integer, dimension(0) :: null_array
    select case (col_in)
    case (0)
      select case (acl_in)
      case (0)
        call color_init_array (col, null_array)
      case default
        call color_init_array (col, (/ -acl_in /))
      end select
    case default
      select case (acl_in)
      case (0)
        call color_init_array (col, (/ col_in /))
      case default
        call color_init_array (col, (/ col_in, -acl_in /))
      end select
    end select
  end subroutine color_init_col_acl

```

This version is used for the external interface. We convert a fixed-size array

of colors (for each particle) to the internal form by packing only the nonzero entries.

```

<Colors: public>+≡
    public :: color_init_from_array

<Colors: interfaces>+≡
    interface color_init_from_array
        module procedure color_init_from_array1, color_init_from_array1g
        module procedure color_init_from_array2, color_init_from_array2g
    end interface

<Colors: procedures>+≡
    pure subroutine color_init_from_array1 (col, c1)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1
        logical, dimension(size(c1)) :: mask
        mask = c1 /= 0
    !   allocate (col%c1 (count (mask)))
    !   allocate (col%c2 (size (col%c1)))
        col%c1 = pack (c1, mask, col%c1)
        col%c2 = col%c1
    end subroutine color_init_from_array1

    pure subroutine color_init_from_array1g (col, c1, ghost)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1
        logical, intent(in) :: ghost
        call color_init_from_array1 (col, c1)
        col%ghost = ghost
    end subroutine color_init_from_array1g

    pure subroutine color_init_from_array2 (col, c1)
        integer, dimension(:, :), intent(in) :: c1
        type(color_t), dimension(size(c1,2)), intent(out) :: col
        integer :: i
        do i = 1, size (c1,2)
            call color_init_from_array1 (col(i), c1(:,i))
        end do
    end subroutine color_init_from_array2

    pure subroutine color_init_from_array2g (col, c1, ghost)
        integer, dimension(:, :), intent(in) :: c1
        type(color_t), dimension(size(c1,2)), intent(out) :: col
        logical, intent(in), dimension(:) :: ghost
        call color_init_from_array2 (col, c1)
        col%ghost = ghost
    end subroutine color_init_from_array2g

```

Set the ghost property

```

<Colors: public>+≡
    public :: color_set_ghost

<Colors: procedures>+≡
    elemental subroutine color_set_ghost (col, ghost)

```

```

    type(color_t), intent(inout) :: col
    logical, intent(in) :: ghost
    col%ghost = ghost
end subroutine color_set_ghost

```

Undefine the color state:

```

<Colors: public>+≡
    public :: color_undefine

<Colors: procedures>+≡
    elemental subroutine color_undefine (col, undefine_ghost)
        type(color_t), intent(inout) :: col
        logical, intent(in), optional :: undefine_ghost
    !   if (allocated (col%c1)) deallocate (col%c1)
    !   if (allocated (col%c2)) deallocate (col%c2)
        col%c1 = 0
        col%c2 = 0
        if (present (undefine_ghost)) then
            if (undefine_ghost) col%ghost = .false.
        else
            col%ghost = .false.
        end if
    end subroutine color_undefine

```

Output. As dense as possible, no linebreak. If color is undefined, no output.

```

<Colors: public>+≡
    public :: color_write

<Colors: interfaces>+≡
    interface color_write
        module procedure color_write_single
        module procedure color_write_array
    end interface

<Colors: procedures>+≡
    subroutine color_write_single (col, unit)
        type(color_t), intent(in) :: col
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        if (color_is_defined (col)) then
            write (u, "(A)", advance="no") "c("
            if (col%c1(1) /= 0) write (u, "(I0)", advance="no") col%c1(1)
            if (any (col%c1 /= 0)) write (u, "(1x)", advance="no")
            if (col%c1(2) /= 0) write (u, "(I0)", advance="no") col%c1(2)
            if (.not. color_is_diagonal (col)) then
                write (u, "(A)", advance="no") "|"
                if (col%c2(1) /= 0) write (u, "(I0)", advance="no") col%c2(1)
                if (any (col%c2 /= 0)) write (u, "(1x)", advance="no")
                if (col%c2(2) /= 0) write (u, "(I0)", advance="no") col%c2(2)
            end if
            write (u, "(A)", advance="no") ")"
        else if (col%ghost) then
            write (u, "(A)", advance="no") "c*"

```

```

        end if
    end subroutine color_write_single

    subroutine color_write_array (col, unit)
        type(color_t), dimension(:), intent(in) :: col
        integer, intent(in), optional :: unit
        integer :: u
        integer :: i
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)", advance="no") "["
        do i = 1, size (col)
            if (i > 1) write (u, "(1x)", advance="no")
            call color_write_single (col(i), u)
        end do
        write (u, "(A)", advance="no") "]"
    end subroutine color_write_array

```

Binary I/O. For allocatable colors, this would have to be modified.

```

<Colors: public>+≡
    public :: color_write_raw
    public :: color_read_raw

<Colors: procedures>+≡
    subroutine color_write_raw (col, u)
        type(color_t), intent(in) :: col
        integer, intent(in) :: u
        logical :: defined
        defined = color_is_defined (col) .or. color_is_ghost (col)
        write (u) defined
        if (defined) then
            write (u) col%c1, col%c2
            write (u) col%ghost
        end if
    end subroutine color_write_raw

    subroutine color_read_raw (col, u, iostat)
        type(color_t), intent(out) :: col
        integer, intent(in) :: u
        integer, intent(out), optional :: iostat
        logical :: defined
        read (u, iostat=iostat) defined
        if (defined) then
            read (u, iostat=iostat) col%c1, col%c2
            read (u, iostat=iostat) col%ghost
        end if
    end subroutine color_read_raw

```

## 7.2.2 Predicates

Return the definition status

```

<Colors: public>+≡
    public :: color_is_defined

```



```

<Colors: procedures>+≡
    elemental function color_is_defined (col) result (defined)
        logical :: defined
        type(color_t), intent(in) :: col
        !    defined = allocated (col%c1)
        defined = any (col%c1 /= 0)
    end function color_is_defined

```

Diagonal color objects have only one array allocated:

```

<Colors: public>+≡
    public :: color_is_diagonal

<Colors: procedures>+≡
    elemental function color_is_diagonal (col) result (diagonal)
        logical :: diagonal
        type(color_t), intent(in) :: col
        if (color_is_defined (col)) then
            diagonal = all (col%c1 == col%c2)
        else
            diagonal = .true.
        end if
    end function color_is_diagonal

```

Return the ghost flag

```

<Colors: public>+≡
    public :: color_is_ghost

<Colors: procedures>+≡
    elemental function color_is_ghost (col) result (ghost)
        logical :: ghost
        type(color_t), intent(in) :: col
        ghost = col%ghost
    end function color_is_ghost

```

The ghost parity: true if the color-ghost flag is set.

```

<Colors: interfaces>+≡
    interface color_ghost_parity
        module procedure color_ghost_parity0
        module procedure color_ghost_parity1
    end interface

<Colors: procedures>+≡
    pure function color_ghost_parity0 (col) result (parity)
        type(color_t), intent(in) :: col
        logical :: parity
        parity = color_is_ghost (col)
    end function color_ghost_parity0

    pure function color_ghost_parity1 (col) result (parity)
        type(color_t), dimension(:), intent(in) :: col
        logical :: parity
        logical, dimension(size(col)) :: p
        integer :: i
        forall (i = 1:size(col))

```

```

        p(i) = color_ghost_parity0 (col(i))
    end forall
    parity = mod (count (p), 2) == 1
end function color_ghost_parity1

```

### 7.2.3 Accessing contents

Return the number of color indices. We assume that it is identical for both arrays.

```

<Colors: procedures>+≡
    elemental function color_number (col) result (n)
        integer :: n
        type(color_t), intent(in) :: col
    !   n = size (col%c1)
        n = count (col%c1 /= 0)
    end function color_number

```

Return the (first) color/anticolor entry (assuming that color is diagonal). The result is a positive color index.

```

<Colors: public>+≡
    public :: color_get_col
    public :: color_get_acl

<Colors: procedures>+≡
    function color_get_col (col) result (c)
        integer :: c
        type(color_t), intent(in) :: col
        integer :: i
        do i = 1, size (col%c1)
            if (col%c1(i) > 0) then
                c = col%c1(i)
                return
            end if
        end do
        c = 0
    end function color_get_col

    function color_get_acl (col) result (c)
        integer :: c
        type(color_t), intent(in) :: col
        integer :: i
        do i = 1, size (col%c1)
            if (col%c1(i) < 0) then
                c = - col%c1(i)
                return
            end if
        end do
        c = 0
    end function color_get_acl

```

Return the color index with highest absolute value

```
(Colors: public)+≡
  public :: color_get_max_value
  !!! ifort 11.1 rev5 bug
  public :: color_get_max_value0
  public :: color_get_max_value1
  public :: color_get_max_value2

(Colors: interfaces)+≡
  interface color_get_max_value
    module procedure color_get_max_value0
    module procedure color_get_max_value1
    module procedure color_get_max_value2
  end interface

(Colors: procedures)+≡
  elemental function color_get_max_value0 (col) result (cmax)
    integer :: cmax
    type(color_t), intent(in) :: col
    cmax = maxval (abs (col%c1))
  end function color_get_max_value0

  pure function color_get_max_value1 (col) result (cmax)
    integer :: cmax
    type(color_t), dimension(:), intent(in) :: col
    cmax = maxval (color_get_max_value0 (col))
  end function color_get_max_value1

  function color_get_max_value2 (col) result (cmax)
    integer :: cmax
    type(color_t), dimension(:,:), intent(in) :: col
    integer, dimension(size(col, 2)) :: cm
    integer :: i
    forall (i = 1:size(col, 2))
      cm(i) = color_get_max_value1 (col(:,i))
    end forall
    cmax = maxval (cm)
  end function color_get_max_value2
```

## 7.2.4 Comparisons

Similar to helicities, colors match if they are equal, or if either one is undefined.

```
(Colors: public)+≡
  public :: operator(.match.)
  public :: operator(==)
  public :: operator(/=)

(Colors: interfaces)+≡
  interface operator(.match.)
    module procedure color_match
  end interface
  interface operator(==)
    module procedure color_eq
```

```

end interface
interface operator(/=)
    module procedure color_neq
end interface

<Colors: procedures>+≡
    elemental function color_match (col1, col2) result (eq)
        logical :: eq
        type(color_t), intent(in) :: col1, col2
        if (color_is_defined (col1) .and. color_is_defined (col2)) then
!           if (size (col1%c1) == size (col2%c1)) then
!               eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2)
!           else
!               eq = .false.
!           end if
        else
            eq = .true.
        end if
    end function color_match

    elemental function color_eq (col1, col2) result (eq)
        logical :: eq
        type(color_t), intent(in) :: col1, col2
        if (color_is_defined (col1) .and. color_is_defined (col2)) then
!           if (size (col1%c1) == size (col2%c1)) then
!               eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2) &
!                   .and. (col1%ghost .eqv. col2%ghost)
!           else
!               eq = .false.
!           end if
        else if (.not. color_is_defined (col1) &
            .and. .not. color_is_defined (col2)) then
            eq = col1%ghost .eqv. col2%ghost
        else
            eq = .false.
        end if
    end function color_eq

    elemental function color_neq (col1, col2) result (neq)
        logical :: neq
        type(color_t), intent(in) :: col1, col2
        if (color_is_defined (col1) .and. color_is_defined (col2)) then
!           if (size (col1%c1) == size (col2%c1)) then
!               neq = any (col1%c1 /= col2%c1) .or. any (col1%c2 /= col2%c2) &
!                   .or. (col1%ghost .neqv. col2%ghost)
!           else
!               neq = .true.
!           end if
        else if (.not. color_is_defined (col1) &
            .and. .not. color_is_defined (col2)) then
            neq = col1%ghost .neqv. col2%ghost
        else
            neq = .true.
        end if
    end function color_neq

```

```

    end if
end function color_neq

```

### 7.2.5 Tools

Shift color indices by a common offset.

```

<Colors: public>+≡
    public :: color_add_offset

<Colors: procedures>+≡
    elemental subroutine color_add_offset (col, offset)
        type(color_t), intent(inout) :: col
        integer, intent(in) :: offset
        where (col%c1 /= 0) col%c1 = col%c1 + sign (offset, col%c1)
        where (col%c2 /= 0) col%c2 = col%c2 + sign (offset, col%c2)
!       if (allocated (col%c1)) then
!           col%c1 = col%c1 + sign (offset, col%c1)
!           col%c2 = col%c2 + sign (offset, col%c2)
!       end if
    end subroutine color_add_offset

```

Reassign color indices for an array of colored particle in canonical order. The allocated size of the color map is such that two colors per particle can be accommodated.

```

<Colors: public>+≡
    public :: color_canonicalize

<Colors: procedures>+≡
    subroutine color_canonicalize (col)
        type(color_t), dimension(:), intent(inout) :: col
        integer, dimension(2*size(col)) :: map
        integer :: n_col, i, j, k
        n_col = 0
        do i = 1, size (col)
            do j = 1, size (col(i)%c1)
                if (col(i)%c1(j) /= 0) then
                    k = find (abs (col(i)%c1(j)), map(:n_col))
                    if (k == 0) then
                        n_col = n_col + 1
                        map(n_col) = abs (col(i)%c1(j))
                        k = n_col
                    end if
                    col(i)%c1(j) = sign (k, col(i)%c1(j))
                end if
                if (col(i)%c2(j) /= 0) then
                    k = find (abs (col(i)%c2(j)), map(:n_col))
                    if (k == 0) then
                        n_col = n_col + 1
                        map(n_col) = abs (col(i)%c2(j))
                        k = n_col
                    end if
                    col(i)%c2(j) = sign (k, col(i)%c2(j))
                end if
            end do
        end do
    end subroutine color_canonicalize

```

```

        end if
    end do
end do
contains
function find (c, array) result (k)
    integer :: k
    integer, intent(in) :: c
    integer, dimension(:), intent(in) :: array
    integer :: i
    k = 0
    do i = 1, size (array)
        if (c == array (i)) then
            k = i
            return
        end if
    end do
end function find
end subroutine color_canonicalize

```

Return an array of different color indices from an array of colors. The last argument is a pseudo-color array, where the color entries correspond to the position of the corresponding index entry in the index array. The colors are assumed to be diagonal.

*(Colors: procedures)+≡*

```

subroutine extract_color_line_indices (col, c_index, col_pos)
    type(color_t), dimension(:), intent(in) :: col
    integer, dimension(:), intent(out), allocatable :: c_index
    type(color_t), dimension(size(col)), intent(out) :: col_pos
    integer, dimension(:), allocatable :: c_tmp
    integer :: i, j, k, n, c
    allocate (c_tmp (sum (color_number (col))))
    n = 0
    SCAN1: do i = 1, size (col)
        SCAN2: do j = 1, 2
            c = abs (col(i)%c1(j))
            if (c /= 0) then
                do k = 1, n
                    if (c_tmp(k) == c) then
                        col_pos(i)%c1(j) = k
                        cycle SCAN2
                    end if
                end do
                n = n + 1
                c_tmp(n) = c
                col_pos(i)%c1(j) = n
            end if
        end do SCAN2
    end do SCAN1
    allocate (c_index (n))
    c_index = c_tmp(1:n)
end subroutine extract_color_line_indices

```

Given a color array, pairwise contract the color lines in all possible ways and

return the resulting array of arrays. The input color array must be diagonal, and each color should occur exactly twice, once as color and once as anticolor.

Gluon entries with equal color and anticolor are explicitly excluded.

This algorithm is generic, but for long arrays it is neither efficient, nor does it avoid duplicates. It is intended for small arrays, in particular for the state matrix of a structure-function pair.

```

<Colors: public>+≡
  public :: color_array_make_contractions

<Colors: procedures>+≡
  subroutine color_array_make_contractions (col_in, col_out)
    type(color_t), dimension(:), intent(in) :: col_in
    type(color_t), dimension(:,:), intent(out), allocatable :: col_out
    type :: entry_t
      integer, dimension(:), allocatable :: map
      type(color_t), dimension(:), allocatable :: col
      type(entry_t), pointer :: next => null ()
    end type entry_t
    type :: list_t
      integer :: n = 0
      type(entry_t), pointer :: first => null ()
      type(entry_t), pointer :: last => null ()
    end type list_t
    type(list_t) :: list
    type(entry_t), pointer :: entry
    integer, dimension(:), allocatable :: c_index
    type(color_t), dimension(size(col_in)) :: col_pos
    integer :: n_prt, n_c_index
    integer, dimension(:), allocatable :: map
    integer :: i, j, c
    n_prt = size (col_in)
    call extract_color_line_indices (col_in, c_index, col_pos)
    ! print *, c_index
    n_c_index = size (c_index)
    allocate (map (n_c_index))
    map = 0
    call list_append_if_valid (list, map)
    entry => list%first
    do while (associated (entry))
      do i = 1, n_c_index
        if (entry%map(i) == 0) then
          c = c_index(i)
          do j = i + 1, n_c_index
            if (entry%map(j) == 0) then
              map = entry%map
              map(i) = c
              map(j) = c
              call list_append_if_valid (list, map)
            end if
          end do
        end if
      end do
      entry => entry%next
    end do
  end subroutine

```

```

    call list_to_array (list, col_out)
contains
  subroutine list_append_if_valid (list, map)
    type(list_t), intent(inout) :: list
    integer, dimension(:), intent(in) :: map
    type(entry_t), pointer :: entry
    integer :: i, j, c, p
    entry => list%first
    do while (associated (entry))
      if (all (map == entry%map)) return
      entry => entry%next
    end do
    allocate (entry)
    allocate (entry%map (n_c_index))
    entry%map = map
    allocate (entry%col (n_prt))
    do i = 1, n_prt
      do j = 1, 2
        c = col_in(i)%c1(j)
        if (c /= 0) then
          p = col_pos(i)%c1(j)
          if (map(p) /= 0) then
            entry%col(i)%c1(j) = sign (map(p), c)
          else
            entry%col(i)%c1(j) = c
          endif
          entry%col(i)%c2(j) = entry%col(i)%c1(j)
        end if
      end do
      if (any (entry%col(i)%c1 /= 0) .and. &
        entry%col(i)%c1(1) == - entry%col(i)%c1(2)) return
    end do
    ! call color_write (entry%col); print *, map
    if (associated (list%last)) then
      list%last%next => entry
    else
      list%first => entry
    end if
    list%last => entry
    list%n = list%n + 1
  end subroutine list_append_if_valid
  subroutine list_to_array (list, col)
    type(list_t), intent(inout) :: list
    type(color_t), dimension(:, :), intent(out), allocatable :: col
    type(entry_t), pointer :: entry
    integer :: i
    allocate (col (n_prt, list%n - 1))
    do i = 0, list%n - 1
      entry => list%first
      list%first => list%first%next
      if (i /= 0) col(:, i) = entry%col
      deallocate (entry)
    end do
    list%last => null ()
  end subroutine list_to_array
end

```



```

        end subroutine list_to_array
    end subroutine color_array_make_contractions

```

Invert the color index, switching from particle to antiparticle. For gluons, we have to swap the order of color entries.

```

<Colors: public>+≡
    public :: color_invert

<Colors: procedures>+≡
    elemental subroutine color_invert (col)
        type(color_t), intent(inout) :: col
        col%c1 = - col%c1
        col%c2 = - col%c2
        if (col%c1(1) < 0 .and. col%c1(2) > 0) then
            col%c1 = col%c1(2:1:-1)
            col%c2 = col%c2(2:1:-1)
        end if
    end subroutine color_invert

```

Make a color map for two matching color arrays. The result is an array of integer pairs.

```

<Colors: public>+≡
    public :: set_color_map

<Colors: procedures>+≡
    subroutine set_color_map (map, col1, col2)
        integer, dimension(:,,:), intent(out), allocatable :: map
        type(color_t), dimension(:), intent(in) :: col1, col2
        integer, dimension(:,,:), allocatable :: map1
        integer :: i, j, k
        allocate (map1 (2, 2 * sum (color_number (col1))))
        k = 0
        do i = 1, size (col1)
            do j = 1, size (col1(i)%c1)
                if (col1(i)%c1(j) /= 0 &
                    .and. all (map1(1,:k) /= abs (col1(i)%c1(j)))) then
                    k = k + 1
                    map1(1,k) = abs (col1(i)%c1(j))
                    map1(2,k) = abs (col2(i)%c1(j))
                end if
                if (col1(i)%c2(j) /= 0 &
                    .and. all (map1(1,:k) /= abs (col1(i)%c2(j)))) then
                    k = k + 1
                    map1(1,k) = abs (col1(i)%c2(j))
                    map1(2,k) = abs (col2(i)%c2(j))
                end if
            end do
        end do
        allocate (map (2, k))
        map(:, :) = map1(:, :k)
    end subroutine set_color_map

```

Translate colors which have a match in the translation table (an array of integer pairs). Color that do not match an entry are simply transferred; this is done by first transferring all components, then modifying entries where appropriate.

```

<Colors: public>+≡
    public :: color_translate

<Colors: interfaces>+≡
    interface color_translate
        module procedure color_translate0
        module procedure color_translate0_offset
        module procedure color_translate1
    end interface

<Colors: procedures>+≡
    subroutine color_translate0 (col, map)
        type(color_t), intent(inout) :: col
        integer, dimension(:, :), intent(in) :: map
        type(color_t) :: col_tmp
        integer :: i
        col_tmp = col
        do i = 1, size (map,2)
            where (abs (col%c1) == map(1,i))
                col_tmp%c1 = sign (map(2,i), col%c1)
            end where
            where (abs (col%c2) == map(1,i))
                col_tmp%c2 = sign (map(2,i), col%c2)
            end where
        end do
        col = col_tmp
    end subroutine color_translate0

    subroutine color_translate0_offset (col, map, offset)
        type(color_t), intent(inout) :: col
        integer, dimension(:, :), intent(in) :: map
        integer, intent(in) :: offset
        logical, dimension(size(col%c1)) :: mask1, mask2
        type(color_t) :: col_tmp
        integer :: i
        col_tmp = col
        mask1 = col%c1 /= 0
        mask2 = col%c2 /= 0
        do i = 1, size (map,2)
            where (abs (col%c1) == map(1,i))
                col_tmp%c1 = sign (map(2,i), col%c1)
                mask1 = .false.
            end where
            where (abs (col%c2) == map(1,i))
                col_tmp%c2 = sign (map(2,i), col%c2)
                mask2 = .false.
            end where
        end do
        col = col_tmp
        where (mask1) col%c1 = sign (abs (col%c1) + offset, col%c1)
        where (mask2) col%c2 = sign (abs (col%c2) + offset, col%c2)
    end subroutine color_translate0_offset

```

```

end subroutine color_translate0_offset

subroutine color_translate1 (col, map, offset)
  type(color_t), dimension(:), intent(inout) :: col
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  integer :: i
  if (present (offset)) then
    do i = 1, size (col)
      call color_translate0_offset (col(i), map, offset)
    end do
  else
    do i = 1, size (col)
      call color_translate0 (col(i), map)
    end do
  end if
end subroutine color_translate1

```

Merge two color objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input colors are defined (and diagonal). If either one is undefined, transfer the defined one.

For a color ghost, color is not defined. These have to be treated separately.

```

<Colors: public>+≡
  public :: operator(.merge.)

<Colors: interfaces>+≡
  interface operator(.merge.)
    module procedure merge_colors
  end interface

<Colors: procedures>+≡
  elemental function merge_colors (col1, col2) result (col)
    type(color_t) :: col
    type(color_t), intent(in) :: col1, col2
    if (color_is_defined (col1) .and. color_is_defined (col2)) then
      call color_init_arrays (col, col1%c1, col2%c1)
    else if (color_is_defined (col1)) then
      col = col1
    else if (color_is_defined (col2)) then
      col = col2
    else if (color_is_ghost (col1)) then
      col = col1
    else if (color_is_ghost (col2)) then
      col = col2
    end if
  end function merge_colors

```

Compute the color factor, given two interfering color arrays.

```

<Colors: public>+≡
  public :: compute_color_factor

<Colors: procedures>+≡
  function compute_color_factor (col1, col2, nc) result (factor)
    real(default) :: factor

```

```

type(color_t), dimension(:), intent(in) :: col1, col2
integer, intent(in), optional :: nc
type(color_t), dimension(size(col1)) :: col
integer :: ncol, nloops, nghost
ncol = 3; if (present(nc)) ncol = nc
col = col1 .merge. col2
nloops = count_color_loops (col)
nghost = count (color_is_ghost (col))
factor = real (ncol, default) ** (nloops - nghost)
if (color_ghost_parity (col)) factor = - factor
end function compute_color_factor

```

We have a pair of color index arrays which corresponds to a squared matrix element. We want to determine the number of color loops in this square matrix element. So we first copy the colors (stored in a single color array with a pair of color lists in each entry) to a temporary where the color indices are shifted by some offset. We then recursively follow each loop, starting at the first color that has the offset, resetting the first color index to the loop index and each further index to zero as we go. We check that (a) each color index occurs twice within the left (right) color array, (b) the loops are closed, so we always come back to a line which has the loop index.

In order for the algorithm to work we have to conjugate the colors of initial state particles (one for decays, two for scatterings) into their corresponding anticolors of outgoing particles.

```

<Colors: public>+≡
public :: count_color_loops

<Colors: procedures>+≡
function count_color_loops (col) result (count)
integer :: count
type(color_t), dimension(:), intent(in) :: col
type(color_t), dimension(size(col)) :: cc
integer :: i, n, offset
!   print *, "Count color loops:"
!   call color_write (col); print *
cc = col
n = size (cc)
offset = n
call color_add_offset (cc, offset)
!   print *, offset
!   call color_write (cc); print *
count = 0
SCAN_LOOPS: do
do i = 1, n
!       print *, i, ':', cc(i)%c1
if (color_is_defined (cc(i))) then
if (any (cc(i)%c1 > offset)) then
!           print *, 'start', i
count = count + 1
call follow_line1 (pick_new_line (cc(i)%c1, count, 1))
cycle SCAN_LOOPS
end if
end if
end if
end if

```

```

        end do
        exit SCAN_LOOPS
    end do SCAN_LOOPS
contains
    function pick_new_line (c, reset_val, sgn) result (line)
        integer :: line
        integer, dimension(:), intent(inout) :: c
        integer, intent(in) :: reset_val
        integer, intent(in) :: sgn
        integer :: i
        if (any (c == count)) then
            line = count
        else
            do i = 1, size (c)
                if (sign (1, c(i)) == sgn .and. abs (c(i)) > offset) then
                    line = c(i)
                    c(i) = reset_val
                    return
                end if
            end do
            call color_mismatch
        end if
    end function pick_new_line
    subroutine reset_line (c, line)
        integer, dimension(:), intent(inout) :: c
        integer, intent(in) :: line
        integer :: i
        do i = 1, size (c)
            if (c(i) == line) then
                c(i) = 0
                return
            end if
        end do
    end subroutine reset_line
    recursive subroutine follow_line1 (line)
        integer, intent(in) :: line
        integer :: i
        ! print *, 'follow line 1:', line
        if (line == count) then
            ! print *, 'loop closed'
            return
        end if
        do i = 1, n
            if (any (cc(i)%c1 == -line)) then
                call reset_line (cc(i)%c1, -line)
                ! print *, 'found', -line, ' resetting c1:'
                ! call color_write (cc); print *
                call follow_line2 (pick_new_line (cc(i)%c2, 0, sign (1, -line)))
                return
            end if
        end do
        call color_mismatch ()
    end subroutine follow_line1
    recursive subroutine follow_line2 (line)

```

```

integer, intent(in) :: line
integer :: i
! print *, 'follow line 2:', line
do i = 1, n
  if (any (cc(i)%c2 == -line)) then
    call reset_line (cc(i)%c2, -line)
!     print *, 'found', -line, ' resetting c2:'
!     call color_write (cc); print *
    call follow_line1 (pick_new_line (cc(i)%c1, 0, sign (1, -line)))
    return
  end if
end do
call color_mismatch ()
end subroutine follow_line2
subroutine color_mismatch ()
  call color_write (col)
  print *
  call msg_bug (" Color flow mismatch (color loops should be closed)")
end subroutine color_mismatch
end function count_color_loops

```

## 7.2.6 Color counting test

*<Colors: public>+≡*

public :: color\_test

*<Colors: procedures>+≡*

```

subroutine color_test ()
  type(color_t), dimension(4) :: col1, col2, col
  type(color_t), dimension(:), allocatable :: col3
  type(color_t), dimension(:,:), allocatable :: col_array
  integer :: count, i
  call color_init_col_acl (col1, (/ 1, 0, 2, 3 /), (/ 0, 1, 3, 2 /))
  col2 = col1
  call color_write (col1); print *
  call color_write (col2); print *
  col = col1 .merge. col2
  call color_write (col); print *
  count = count_color_loops (col)
  print *, "Number of color loops (3): ", count
  call color_init_col_acl (col2, (/ 1, 0, 2, 3 /), (/ 0, 2, 3, 1 /))
  call color_write (col1); print *
  call color_write (col2); print *
  col = col1 .merge. col2
  call color_write (col); print *
  count = count_color_loops (col)
  print *, "Number of color loops (2): ", count
  print *
  allocate (col3 (4))
  call color_init_from_array (col3, &
    reshape ((/ 1, 0, 0, -1, 2, -3, 3, -2 /), &
      (/ 2, 4 /)))
  call color_write (col3); print *

```

```

call color_array_make_contractions (col3, col_array)
print *, "Contractions:"
do i = 1, size (col_array, 2)
    call color_write (col_array(:,i)); print *
end do
deallocate (col3)
print *
allocate (col3 (6))
call color_init_from_array (col3, &
    reshape ((/ 1, -2, 3, 0, 0, -1, 2, -4, -3, 0, 4, 0 /), &
        (/ 2, 6 /)))
call color_write (col3); print *
call color_array_make_contractions (col3, col_array)
print *, "Contractions:"
do i = 1, size (col_array, 2)
    call color_write (col_array(:,i)); print *
end do
end subroutine color_test

```

### 7.2.7 The Madgraph color model

This section describes the method for matrix element and color flow calculation within Madgraph.

For each Feynman diagram, the colorless amplitude for a specified helicity and momentum configuration (in- and out- combined) is computed:

$$A_d(p, h) \quad (7.1)$$

Inserting color, the squared matrix element for definite helicity and momentum is

$$M^2(p, h) = \sum_{dd'} A_d(p, h) C_{dd'} A_{d'}^*(p, h) \quad (7.2)$$

where  $C_{dd'}$  describes the color interference of the two diagrams  $A_d$  and  $A_{d'}$ , which is independent of momentum and helicity and can be calculated for each Feynman diagram pair by reducing it to the corresponding color graph. Obviously, one could combine all diagrams with identical color structure, such that the index  $d$  runs only over different color graphs. For colorless diagrams all elements of  $C_{dd'}$  are equal to unity.

The hermitian matrix  $C_{dd'}$  is diagonalized once and for all, such that it can be written in the form

$$C_{dd'} = \sum_{\lambda} c_d^{\lambda} \lambda c_{d'}^{\lambda*}, \quad (7.3)$$

where the eigenvectors  $c_d$  are normalized,

$$\sum_d |c_d^{\lambda}|^2 = 1, \quad (7.4)$$

and the  $\lambda$  values are the corresponding eigenvalues. In the colorless case, this means  $c_d = 1/\sqrt{N_d}$  for all diagrams ( $N_d$  = number of diagrams), and  $\lambda = N_d$  is the only nonzero eigenvalue.

Consequently, the squared matrix element for definite helicity and momentum can also be written as

$$M^2(p, h) = \sum_{\lambda} A_{\lambda}(p, h) \lambda A_{\lambda}(p, h)^* \quad (7.5)$$

with

$$A_{\lambda}(p, h) = \sum_d c_d^{\lambda} A_d(p, h). \quad (7.6)$$

For generic spin density matrices, this is easily generalized to

$$M^2(p, h, h') = \sum_{\lambda} A_{\lambda}(p, h) \lambda A_{\lambda}(p, h')^* \quad (7.7)$$

To determine the color flow probabilities of a given momentum-helicity configuration, the color flow amplitudes are calculated as

$$a_f(p, h) = \sum_d \beta_d^f A_d(p, h), \quad (7.8)$$

where the coefficients  $\beta_d^f$  describe the amplitude for a given Feynman diagram (or color graph)  $d$  to correspond to a definite color flow  $f$ . They are computed from  $C_{dd'}$  by transforming this matrix into the color flow basis and neglecting all off-diagonal elements. Again, these coefficients do not depend on momentum or helicity and can therefore be calculated in advance. This gives the color flow transition matrix

$$F^f(p, h, h') = a_f(p, h) a_f^*(p, h') \quad (7.9)$$

which is assumed diagonal in color flow space and is separate from the color-summed transition matrix  $M^2$ . They are, however, equivalent (up to a factor) to leading order in  $1/N_c$ , and using the color flow transition matrix is appropriate for matching to hadronization.

Note that the color flow transition matrix is not normalized at this stage. To make use of it, we have to fold it with the in-state density matrix to get a pseudo density matrix

$$\hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h'_{\text{out}}) = \sum_{h_{\text{in}} h'_{\text{in}}} F^f(p, h, h') \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}) \quad (7.10)$$

which gets a meaning only after contracted with projections on the outgoing helicity states  $k_{\text{out}}$ , given as linear combinations of helicity states with the unitary coefficient matrix  $c(k_{\text{out}}, h_{\text{out}})$ . Then the probability of finding color flow  $f$  when the helicity state  $k_{\text{out}}$  is measured is given by

$$P^f(p, k_{\text{out}}) = Q^f(p, k_{\text{out}}) / \sum_f Q^f(p, k_{\text{out}}) \quad (7.11)$$

where

$$Q^f(p, k_{\text{out}}) = \sum_{h_{\text{out}} h'_{\text{out}}} c(k_{\text{out}}, h_{\text{out}}) \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h'_{\text{out}}) c^*(k_{\text{out}}, h'_{\text{out}}) \quad (7.12)$$



However, if we can assume that the out-state helicity basis is the canonical one, we can throw away the off diagonal elements in the color flow density matrix and normalize the ones on the diagonal to obtain

$$P^f(p, h_{\text{out}}) = \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}) / \sum_f \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}) \quad (7.13)$$

Finally, the color-summed out-state density matrix is computed by the scattering formula

$$\rho_{\text{out}}(p, h_{\text{out}}, h'_{\text{out}}) = \sum_{h_{\text{in}} h'_{\text{in}}} M^2(p, h, h') \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}) \quad (7.14)$$

$$= \sum_{h_{\text{in}} h'_{\text{in}} \lambda} A_{\lambda}(p, h) \lambda A_{\lambda}(p, h')^* \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}), \quad (7.15)$$

The trace of  $\rho_{\text{out}}$  is the squared matrix element, summed over all internal degrees of freedom. To get the squared matrix element for a definite helicity  $k_{\text{out}}$  and color flow  $f$ , one has to project the density matrix onto the given helicity state and multiply with  $P^f(p, k_{\text{out}})$ .

For diagonal helicities the out-state density reduces to

$$\rho_{\text{out}}(p, h_{\text{out}}) = \sum_{h_{\text{in}} \lambda} \lambda |A_{\lambda}(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \quad (7.16)$$

Since no basis transformation is involved, we can use the normalized color flow probability  $P^f(p, h_{\text{out}})$  and express the result as

$$\rho_{\text{out}}^f(p, h_{\text{out}}) = \rho_{\text{out}}(p, h_{\text{out}}) P^f(p, h_{\text{out}}) \quad (7.17)$$

$$= \sum_{h_{\text{in}} \lambda} \frac{|a^f(p, h)|^2}{\sum_f |a^f(p, h)|^2} \lambda |A_{\lambda}(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \quad (7.18)$$

From these considerations, the following calculation strategy can be derived:

- Before the first event is generated, the color interference matrix  $C_{dd'}$  is computed and diagonalized, so the eigenvectors  $c_d^{\lambda}$ , eigenvalues  $\lambda$  and color flow coefficients  $\beta_d^f$  are obtained. In practice, these calculations are done when the matrix element code is generated, and the results are hardcoded in the matrix element subroutine as **DATA** statements.
- For each event, one loops over helicities once and stores the matrices  $A_{\lambda}(p, h)$  and  $a^f(p, h)$ . The allowed color flows, helicity combinations and eigenvalues are each labeled by integer indices, so one has to store complex matrices of dimension  $N_{\lambda} \times N_h$  and  $N_f \times N_h$ , respectively.
- The further strategy depends on the requested information.
  1. If colorless diagonal helicity amplitudes are required, the eigenvalues  $A_{\lambda}(p, h)$  are squared, summed with weight  $\lambda$ , and the result contracted with the in-state probability vector  $\rho_{\text{in}}(p, h_{\text{in}})$ . The result is a probability vector  $\rho_{\text{out}}(p, h_{\text{out}})$ .

2. For colored diagonal helicity amplitudes, the color coefficients  $a^f(p, h)$  are also squared and used as weights to obtain the color-flow probability vector  $\rho_{\text{out}}^f(p, h_{\text{out}})$ .
3. For colorless non-diagonal helicity amplitudes, we contract the tensor product of  $A_\lambda(p, h)$  with  $A_\lambda(p, h')$ , weighted with  $\lambda$ , with the correlated in-state density matrix, to obtain a correlated out-state density matrix.
4. In the general (colored, non-diagonal) case, we do the same as in the colorless case, but return the un-normalized color flow density matrix  $\hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h'_{\text{out}})$  in addition. When the relevant helicity basis is known, the latter can be used by the caller program to determine flow probabilities. (In reality, we assume the canonical basis and reduce the correlated out-state density to its diagonal immediately.)

## 7.3 Flavors: Particle properties

This module contains a type for holding the flavor code, and all functions that depend on the model, i.e., that determine particle properties.

The PDG code is packed in a special `flavor` type. (This prohibits meaningless operations, and it allows for a different implementation, e.g., some non-PDG scheme internally, if appropriate at some point.) In addition, the flavor object holds a pointer to a `particle_data` object which is centrally stored (as part of a physics model). In this way, all particle data can be accessed using just the `flv` object without having to carry them around.

The pointer component imposes a technical restriction: Assignment of flavor objects cannot be used, directly or indirectly, in pure, and thus in elemental procedures.

```

<flavors.f90>≡
  <File header>

  module flavors

    <Use kinds>
    <Use strings>
    <Use file utils>
    use pdg_arrays
    use colors, only: color_t, color_init
    use models

    <Standard module head>

    <Flavors: public>

    <Flavors: types>

    <Flavors: interfaces>

    contains

    <Flavors: procedures>

  end module flavors

```

### 7.3.1 The flavor type

The flavor type is an integer representing the PDG code, or undefined (zero). Negative codes represent antiparticles.

For full generality, and analogy with helicity and color, we allow for non-diagonal flavor indices in density matrices. This is probably academic; an obscure application is the definition of proper isospin states.

Further properties of the given flavor can be retrieved via the particle-data pointer, if it is associated.

```

<Flavors: public>≡
  public :: flavor_t

```

```

<Flavors: types>≡
  type :: flavor_t
  private
  integer :: f = UNDEFINED
  type(particle_data_t), pointer :: prt => null ()
end type flavor_t

```

Initializer form. If the model is assigned, the procedure is impure, therefore we have to define a separate array version.

```

<Flavors: public>+≡
  public :: flavor_init

<Flavors: interfaces>≡
  interface flavor_init
    module procedure flavor_init0_empty
    module procedure flavor_init0
    module procedure flavor_init0_particle_data
    module procedure flavor_init0_model
    module procedure flavor_init0_name_model
    module procedure flavor_init1_model
    module procedure flavor_init1_name_model
    module procedure flavor_init2_model
    module procedure flavor_init_aval_model
  end interface

<Flavors: procedures>≡
  elemental subroutine flavor_init0_empty (flv)
    type(flavor_t), intent(out) :: flv
  end subroutine flavor_init0_empty

  elemental subroutine flavor_init0 (flv, f)
    type(flavor_t), intent(out) :: flv
    integer, intent(in) :: f
    flv%f = f
  end subroutine flavor_init0

  subroutine flavor_init0_particle_data (flv, particle_data)
    type(flavor_t), intent(out) :: flv
    type(particle_data_t), intent(in), target :: particle_data
    flv%f = particle_data_get_pdg (particle_data)
    flv%prt => particle_data
  end subroutine flavor_init0_particle_data

  subroutine flavor_init0_model (flv, f, model)
    type(flavor_t), intent(out) :: flv
    integer, intent(in) :: f
    type(model_t), intent(in), target :: model
    flv%f = f
    flv%prt => model_get_particle_ptr (model, f)
  end subroutine flavor_init0_model

  subroutine flavor_init1_model (flv, f, model)
    type(flavor_t), dimension(:), intent(out) :: flv
    integer, dimension(:), intent(in) :: f
    type(model_t), intent(in), target :: model

```

```

integer :: i
do i = 1, size (f)
    call flavor_init0_model (flv(i), f(i), model)
end do
end subroutine flavor_init1_model

subroutine flavor_init2_model (flv, f, model)
    type(flavor_t), dimension(:,:), intent(out) :: flv
    integer, dimension(:,:), intent(in) :: f
    type(model_t), intent(in), target :: model
    integer :: i
    do i = 1, size (f, 2)
        call flavor_init1_model (flv(:,i), f(:,i), model)
    end do
end subroutine flavor_init2_model

subroutine flavor_init0_name_model (flv, name, model)
    type(flavor_t), intent(out) :: flv
    type(string_t), intent(in) :: name
    type(model_t), intent(in), target :: model
    flv%f = model_get_particle_pdg (model, name)
    flv%prt => model_get_particle_ptr (model, flv%f)
end subroutine flavor_init0_name_model

subroutine flavor_init1_name_model (flv, name, model)
    type(flavor_t), dimension(:), intent(out) :: flv
    type(string_t), dimension(:), intent(in) :: name
    type(model_t), intent(in), target :: model
    integer :: i
    do i = 1, size (name)
        call flavor_init0_name_model (flv(i), name(i), model)
    end do
end subroutine flavor_init1_name_model

```

This version transforms a PDG array value into a flavor array. The flavor array must be allocatable.

```

<Flavors: procedures>+≡
subroutine flavor_init_aval_model (flv, aval, model)
    type(flavor_t), dimension(:), intent(out), allocatable :: flv
    type(pdg_array_t), intent(in) :: aval
    type(model_t), intent(in), target :: model
    integer, dimension(:), allocatable :: pdg
    pdg = aval
    allocate (flv (size (pdg)))
    call flavor_init (flv, pdg, model)
end subroutine flavor_init_aval_model

```

Undefine the flavor state:

```

<Flavors: public>+≡
public :: flavor_undefine

<Flavors: procedures>+≡
elemental subroutine flavor_undefine (flv)

```

```

    type(flavor_t), intent(inout) :: flv
    flv%f = UNDEFINED
    flv%prt => null ()
end subroutine flavor_undefine

```

Output: dense, no linebreak

*<Flavors: public>+≡*

```

public :: flavor_write

```

*<Flavors: procedures>+≡*

```

subroutine flavor_write (flv, unit)
    type(flavor_t), intent(in) :: flv
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    if (associated (flv%prt)) then
        write (u, "(A)", advance="no") "f("
    else
        write (u, "(A)", advance="no") "p("
    end if
    write (u, "(I0)", advance="no") flv%f
    write (u, "(A)", advance="no") ")"
end subroutine flavor_write

```

Binary I/O. Currently, the model information is not written/read, so after reading the particle-data pointer is empty.

*<Flavors: public>+≡*

```

public :: flavor_write_raw
public :: flavor_read_raw

```

*<Flavors: procedures>+≡*

```

subroutine flavor_write_raw (flv, u)
    type(flavor_t), intent(in) :: flv
    integer, intent(in) :: u
    write (u) flv%f
end subroutine flavor_write_raw

subroutine flavor_read_raw (flv, u, iostat)
    type(flavor_t), intent(out) :: flv
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    read (u, iostat=iostat) flv%f
end subroutine flavor_read_raw

```

## Assignment

Default assignment of flavor objects is possible, but cannot be used in pure procedures, because a pointer assignment is involved.

Assign the particle pointer separately. This cannot be elemental, so we define a scalar and an array version explicitly. We refer to an array of flavors, not an array of models.

*<Flavors: public>+≡*

```

public :: flavor_set_model
<Flavors: interfaces>+≡
interface flavor_set_model
  module procedure flavor_set_model_single
  module procedure flavor_set_model_array
end interface
<Flavors: procedures>+≡
subroutine flavor_set_model_single (flv, model)
  type(flavor_t), intent(inout) :: flv
  type(model_t), intent(in), target :: model
  if (flv%f /= UNDEFINED) &
    flv%prt => model_get_particle_ptr (model, flv%f)
end subroutine flavor_set_model_single

subroutine flavor_set_model_array (flv, model)
  type(flavor_t), dimension(:), intent(inout) :: flv
  type(model_t), intent(in), target :: model
  integer :: i
  do i = 1, size (flv)
    if (flv(i)%f /= UNDEFINED) &
      flv(i)%prt => model_get_particle_ptr (model, flv(i)%f)
    end do
  end do
end subroutine flavor_set_model_array

```

## Predicates

Return the definition status

```

<Flavors: public>+≡
public :: flavor_is_defined
<Flavors: procedures>+≡
elemental function flavor_is_defined (flv) result (defined)
  logical :: defined
  type(flavor_t), intent(in) :: flv
  defined = flv%f /= UNDEFINED
end function flavor_is_defined

```

Check for valid flavor (including undefined):

```

<Flavors: public>+≡
public :: flavor_is_valid
<Flavors: procedures>+≡
elemental function flavor_is_valid (flv) result (valid)
  logical :: valid
  type(flavor_t), intent(in) :: flv
  valid = flv%f /= INVALID
end function flavor_is_valid

```

Return true if the particle-data pointer is associated. (Debugging aid)

```

<Flavors: public>+≡
public :: flavor_is_associated

```

```

<Flavors: procedures>+=
  elemental function flavor_is_associated (flv) result (flag)
    logical :: flag
    type(flavor_t), intent(in) :: flv
    flag = associated (flv%prt)
  end function flavor_is_associated

```

## Accessing contents

With the exception of the PDG code, all particle properties are accessible only via the `prt` pointer. If this is unassigned, some access function will crash.

Return the flavor as an integer

```

<Flavors: public>+=
  public :: flavor_get_pdg

<Flavors: procedures>+=
  elemental function flavor_get_pdg (flv) result (f)
    integer :: f
    type(flavor_t), intent(in) :: flv
    f = flv%f
  end function flavor_get_pdg

```

Return the flavor of the antiparticle

```

<Flavors: public>+=
  public :: flavor_get_pdg_anti

<Flavors: procedures>+=
  elemental function flavor_get_pdg_anti (flv) result (f)
    integer :: f
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
      if (particle_data_has_antiparticle (flv%prt)) then
        f = -flv%f
      else
        f = flv%f
      end if
    else
      f = 0
    end if
  end function flavor_get_pdg_anti

```

Absolute value:

```

<Flavors: public>+=
  public :: flavor_get_pdg_abs

<Flavors: procedures>+=
  elemental function flavor_get_pdg_abs (flv) result (f)
    integer :: f
    type(flavor_t), intent(in) :: flv
    f = abs (flv%f)
  end function flavor_get_pdg_abs

```



Generic properties

*(Flavors: public)*+≡

```
public :: flavor_is_visible
public :: flavor_is_parton
public :: flavor_is_beam_remnant
public :: flavor_is_gauge
public :: flavor_is_left_handed
public :: flavor_is_right_handed
public :: flavor_is_antiparticle
public :: flavor_has_antiparticle
public :: flavor_is_stable
public :: flavor_decays_isotropically
public :: flavor_decays_diagonal
public :: flavor_is_polarized
```

*(Flavors: procedures)*+≡

```
elemental function flavor_is_visible (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
    flag = particle_data_is_visible (flv%prt)
  else
    flag = .false.
  end if
end function flavor_is_visible
```

```
elemental function flavor_is_parton (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
    flag = particle_data_is_parton (flv%prt)
  else
    flag = .false.
  end if
end function flavor_is_parton
```

```
elemental function flavor_is_beam_remnant (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  select case (abs (flv%f))
  case (HADRON_REMNANT, &
        HADRON_REMNANT_SINGLET, HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
    flag = .true.
  case default
    flag = .false.
  end select
end function flavor_is_beam_remnant
```

```
elemental function flavor_is_gauge (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
    flag = particle_data_is_gauge (flv%prt)
  else
    flag = .false.
  end if
end function flavor_is_gauge
```

```

        end if
    end function flavor_is_gauge

    elemental function flavor_is_left_handed (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            if (flv%f > 0) then
                flag = particle_data_is_left_handed (flv%prt)
            else
                flag = particle_data_is_right_handed (flv%prt)
            end if
        else
            flag = .false.
        end if
    end function flavor_is_left_handed

    elemental function flavor_is_right_handed (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            if (flv%f > 0) then
                flag = particle_data_is_right_handed (flv%prt)
            else
                flag = particle_data_is_left_handed (flv%prt)
            end if
        else
            flag = .false.
        end if
    end function flavor_is_right_handed

    elemental function flavor_is_antiparticle (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        flag = flv%f < 0
    end function flavor_is_antiparticle

    elemental function flavor_has_antiparticle (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            flag = particle_data_has_antiparticle (flv%prt)
        else
            flag = .false.
        end if
    end function flavor_has_antiparticle

    elemental function flavor_is_stable (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            flag = particle_data_is_stable (flv%prt, anti = flv%f < 0)
        else
            flag = .true.
        end if
    end function flavor_is_stable

```

```

        end if
    end function flavor_is_stable

    elemental function flavor_decays_isotropically (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            flag = particle_data_decays_isotropically (flv%prt, anti = flv%f < 0)
        else
            flag = .true.
        end if
    end function flavor_decays_isotropically

    elemental function flavor_decays_diagonal (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            flag = particle_data_decays_diagonal (flv%prt, anti = flv%f < 0)
        else
            flag = .true.
        end if
    end function flavor_decays_diagonal

    elemental function flavor_is_polarized (flv) result (flag)
        logical :: flag
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            flag = particle_data_is_polarized (flv%prt, anti = flv%f < 0)
        else
            flag = .false.
        end if
    end function flavor_is_polarized

```

Names:

*(Flavors: public)*+≡

```

    public :: flavor_get_name
    public :: flavor_get_tex_name

```

*(Flavors: procedures)*+≡

```

    elemental function flavor_get_name (flv) result (name)
        type(string_t) :: name
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            name = particle_data_get_name (flv%prt, flv%f < 0)
        else
            name = "?"
        end if
    end function flavor_get_name

    elemental function flavor_get_tex_name (flv) result (name)
        type(string_t) :: name
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            name = particle_data_get_tex_name (flv%prt, flv%f < 0)

```

```

        else
            name = "?"
        end if
    end function flavor_get_tex_name

```

*<Flavors: public>+≡*

```

    public :: flavor_get_spin_type
    public :: flavor_get_multiplicity
    public :: flavor_get_isospin_type
    public :: flavor_get_charge_type
    public :: flavor_get_color_type

```

*<Flavors: procedures>+≡*

```

    elemental function flavor_get_spin_type (flv) result (type)
        integer :: type
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            type = particle_data_get_spin_type (flv%prt)
        else
            type = 1
        end if
    end function flavor_get_spin_type

```

```

    elemental function flavor_get_multiplicity (flv) result (type)
        integer :: type
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            type = particle_data_get_multiplicity (flv%prt)
        else
            type = 1
        end if
    end function flavor_get_multiplicity

```

```

    elemental function flavor_get_isospin_type (flv) result (type)
        integer :: type
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            type = particle_data_get_isospin_type (flv%prt)
        else
            type = 1
        end if
    end function flavor_get_isospin_type

```

```

    elemental function flavor_get_charge_type (flv) result (type)
        integer :: type
        type(flavor_t), intent(in) :: flv
        if (associated (flv%prt)) then
            type = particle_data_get_charge_type (flv%prt)
        else
            type = 1
        end if
    end function flavor_get_charge_type

```

```

    elemental function flavor_get_color_type (flv) result (type)

```

```

integer :: type
type(flavor_t), intent(in) :: flv
if (associated (flv%prt)) then
    if (flavor_is_antiparticle (flv)) then
        type = - particle_data_get_color_type (flv%prt)
    else
        type = particle_data_get_color_type (flv%prt)
    end if
else
    type = 1
end if
end function flavor_get_color_type

```

These functions return real values:

*<Flavors: public>+≡*

```

public :: flavor_get_charge
public :: flavor_get_mass
public :: flavor_get_width
public :: flavor_get_isospin

```

*<Flavors: procedures>+≡*

```

elemental function flavor_get_charge (flv) result (charge)
    real(default) :: charge
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
        if (flavor_is_antiparticle (flv)) then
            charge = particle_data_get_charge (flv%prt)
        else
            charge = - particle_data_get_charge (flv%prt)
        end if
    else
        charge = 0
    end if
end function flavor_get_charge

elemental function flavor_get_mass (flv) result (mass)
    real(default) :: mass
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
        mass = particle_data_get_mass (flv%prt)
    else
        mass = 0
    end if
end function flavor_get_mass

elemental function flavor_get_width (flv) result (width)
    real(default) :: width
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
        width = particle_data_get_width (flv%prt)
    else
        width = 0
    end if
end function flavor_get_width

```

```

elemental function flavor_get_isospin (flv) result (isospin)
  real(default) :: isospin
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
    if (flavor_is_antiparticle (flv)) then
      isospin = particle_data_get_isospin (flv%prt)
    else
      isospin = - particle_data_get_isospin (flv%prt)
    end if
  else
    isospin = 0
  end if
end function flavor_get_isospin

```

## Comparisons

If one of the flavors is undefined, the other defined, they match.

*(Flavors: public)*+≡

```

public :: operator(.match.)
public :: operator(==)
public :: operator(/=)

```

*(Flavors: interfaces)*+≡

```

interface operator(.match.)
  module procedure flavor_match
end interface
interface operator(==)
  module procedure flavor_eq
end interface
interface operator(/=)
  module procedure flavor_neq
end interface

```

*(Flavors: procedures)*+≡

```

elemental function flavor_match (flv1, flv2) result (eq)
  logical :: eq
  type(flavor_t), intent(in) :: flv1, flv2
  if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
    eq = flv1%f == flv2%f
  else
    eq = .true.
  end if
end function flavor_match

elemental function flavor_eq (flv1, flv2) result (eq)
  logical :: eq
  type(flavor_t), intent(in) :: flv1, flv2
  if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
    eq = flv1%f == flv2%f
  else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
    eq = .true.
  else
    eq = .false.
  end if
end function flavor_eq

```

```

    end if
end function flavor_eq

```

```

<Flavors: procedures>+=
elemental function flavor_neq (flv1, flv2) result (neq)
    logical :: neq
    type(flavor_t), intent(in) :: flv1, flv2
    if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
        neq = flv1%f /= flv2%f
    else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
        neq = .false.
    else
        neq = .true.
    end if
end function flavor_neq

```

## Tools

Merge two flavor indices. This works only if both are equal or either one is undefined, because we have no off-diagonal flavor entries. Otherwise, generate an invalid flavor.

We cannot use elemental procedures because of the pointer component.

```

<Flavors: public>+=
public :: operator(.merge.)

<Flavors: interfaces>+=
interface operator(.merge.)
    module procedure merge_flavors0
    module procedure merge_flavors1
end interface

<Flavors: procedures>+=
function merge_flavors0 (flv1, flv2) result (flv)
    type(flavor_t) :: flv
    type(flavor_t), intent(in) :: flv1, flv2
    if (flavor_is_defined (flv1) .and. flavor_is_defined (flv2)) then
        if (flv1 == flv2) then
            flv = flv1
        else
            flv%f = INVALID
        end if
    else if (flavor_is_defined (flv1)) then
        flv = flv1
    else if (flavor_is_defined (flv2)) then
        flv = flv2
    end if
end function merge_flavors0

function merge_flavors1 (flv1, flv2) result (flv)
    type(flavor_t), dimension(:), intent(in) :: flv1, flv2
    type(flavor_t), dimension(size(flv1)) :: flv
    integer :: i

```

```

do i = 1, size (flv1)
  flv(i) = flv1(i) .merge. flv2(i)
end do
end function merge_flavors1

```

Generate consecutive color indices for a given flavor. The indices are counted starting with the stored value of *c*, so new indices are created each time this (impure) function is called. The counter can be reset by the optional argument *c\_seed* if desired. The optional flag *reverse* is used only for octets. If set, the color and anticolor entries of the octet particle are exchanged.

```

<Flavors: public>+≡
  public :: color_from_flavor

<Flavors: interfaces>+≡
  interface color_from_flavor
    module procedure color_from_flavor0
    module procedure color_from_flavor1
  end interface

<Flavors: procedures>+≡
  function color_from_flavor0 (flv, c_seed, reverse) result (col)
    type(color_t) :: col
    type(flavor_t), intent(in) :: flv
    integer, intent(in), optional :: c_seed
    logical, intent(in), optional :: reverse
    integer, save :: c = 1
    logical :: rev
    if (present (c_seed)) c = c_seed
    rev = .false.; if (present (reverse)) rev = reverse
    select case (flavor_get_color_type (flv))
    case (1)
    case (3)
      call color_init (col, (/ c /)); c = c + 1
    case (-3)
      call color_init (col, (/ -c /)); c = c + 1
    case (8)
      if (rev) then
        call color_init (col, (/ c+1, -c /)); c = c + 2
      else
        call color_init (col, (/ c, -(c+1) /)); c = c + 2
      end if
    end select
  end function color_from_flavor0

  function color_from_flavor1 (flv, c_seed, reverse) result (col)
    type(flavor_t), dimension(:), intent(in) :: flv
    integer, intent(in), optional :: c_seed
    logical, intent(in), optional :: reverse
    type(color_t), dimension(size(flv)) :: col
    integer :: i
    col(1) = color_from_flavor0 (flv(1), c_seed, reverse)
    do i = 2, size (flv)
      col(i) = color_from_flavor0 (flv(i), reverse=reverse)
    end do

```



```
end function color_from_flavor1
```

This procedure returns the flavor object for the antiparticle. The antiparticle code may either be the same code or its negative.

*<Flavors: public>+≡*

```
public :: flavor_anti
```

*<Flavors: procedures>+≡*

```
function flavor_anti (flv) result (aflv)
  type(flavor_t) :: aflv
  type(flavor_t), intent(in) :: flv
  if (flavor_has_antiparticle (flv)) then
    aflv%f = - flv%f
  else
    aflv%f = flv%f
  end if
  aflv%prt => flv%prt
end function flavor_anti
```

## 7.4 Quantum numbers

This module collects helicity, color, and flavor in a single type and defines procedures

```
<quantum_numbers.f90>≡  
  <File header>  
  
  module quantum_numbers  
  
    <Use file utils>  
    use models  
    use flavors  
    use colors  
    use helicities  
  
    <Standard module head>  
  
    <Quantum numbers: public>  
  
    <Quantum numbers: types>  
  
    <Quantum numbers: interfaces>  
  
    contains  
  
    <Quantum numbers: procedures>  
  
  end module quantum_numbers
```

### 7.4.1 The quantum number type

```
<Quantum numbers: public>≡  
  public :: quantum_numbers_t  
  
<Quantum numbers: types>≡  
  type :: quantum_numbers_t  
    private  
    type(flavor_t) :: f  
    type(color_t) :: c  
    type(helicity_t) :: h  
  end type quantum_numbers_t
```

Define quantum numbers: Initializer form. All arguments may be present or absent.

```
<Quantum numbers: public>+≡  
  public :: quantum_numbers_init  
  
<Quantum numbers: interfaces>≡  
  interface quantum_numbers_init  
    module procedure quantum_numbers_init0_f  
    module procedure quantum_numbers_init0_c  
    module procedure quantum_numbers_init0_h  
    module procedure quantum_numbers_init0_fc  
    module procedure quantum_numbers_init0_fh
```

```

module procedure quantum_numbers_init0_ch
module procedure quantum_numbers_init0_fch
module procedure quantum_numbers_init1_f
module procedure quantum_numbers_init1_c
module procedure quantum_numbers_init1_h
module procedure quantum_numbers_init1_fc
module procedure quantum_numbers_init1_fh
module procedure quantum_numbers_init1_ch
module procedure quantum_numbers_init1_fch
end interface

```

*(Quantum numbers: procedures)*≡

```

subroutine quantum_numbers_init0_f (qn, flv)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  qn%f = flv
end subroutine quantum_numbers_init0_f

subroutine quantum_numbers_init0_c (qn, col)
  type(quantum_numbers_t), intent(out) :: qn
  type(color_t), intent(in) :: col
  qn%c = col
end subroutine quantum_numbers_init0_c

subroutine quantum_numbers_init0_h (qn, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(helicity_t), intent(in) :: hel
  qn%h = hel
end subroutine quantum_numbers_init0_h

subroutine quantum_numbers_init0_fc (qn, flv, col)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  qn%f = flv
  qn%c = col
end subroutine quantum_numbers_init0_fc

subroutine quantum_numbers_init0_fh (qn, flv, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  qn%h = hel
end subroutine quantum_numbers_init0_fh

subroutine quantum_numbers_init0_ch (qn, col, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  qn%c = col
  qn%h = hel
end subroutine quantum_numbers_init0_ch

```

```

subroutine quantum_numbers_init0_fch (qn, flv, col, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  qn%c = col
  qn%h = hel
end subroutine quantum_numbers_init0_fch

subroutine quantum_numbers_init1_f (qn, flv)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: i
  do i = 1, size (qn)
    call quantum_numbers_init0_f (qn(i), flv(i))
  end do
end subroutine quantum_numbers_init1_f

subroutine quantum_numbers_init1_c (qn, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
    call quantum_numbers_init0_c (qn(i), col(i))
  end do
end subroutine quantum_numbers_init1_c

subroutine quantum_numbers_init1_h (qn, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
    call quantum_numbers_init0_h (qn(i), hel(i))
  end do
end subroutine quantum_numbers_init1_h

subroutine quantum_numbers_init1_fc (qn, flv, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
    call quantum_numbers_init0_fc (qn(i), flv(i), col(i))
  end do
end subroutine quantum_numbers_init1_fc

subroutine quantum_numbers_init1_fh (qn, flv, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
    call quantum_numbers_init0_fh (qn(i), flv(i), hel(i))
  end do
end subroutine quantum_numbers_init1_fh

```

```

        end do
    end subroutine quantum_numbers_init1_fh

    subroutine quantum_numbers_init1_ch (qn, col, hel)
        type(quantum_numbers_t), dimension(:), intent(out) :: qn
        type(color_t), dimension(:), intent(in) :: col
        type(helicity_t), dimension(:), intent(in) :: hel
        integer :: i
        do i = 1, size (qn)
            call quantum_numbers_init0_ch (qn(i), col(i), hel(i))
        end do
    end subroutine quantum_numbers_init1_ch

    subroutine quantum_numbers_init1_fch (qn, flv, col, hel)
        type(quantum_numbers_t), dimension(:), intent(out) :: qn
        type(flavor_t), dimension(:), intent(in) :: flv
        type(color_t), dimension(:), intent(in) :: col
        type(helicity_t), dimension(:), intent(in) :: hel
        integer :: i
        do i = 1, size (qn)
            call quantum_numbers_init0_fch (qn(i), flv(i), col(i), hel(i))
        end do
    end subroutine quantum_numbers_init1_fch

```

### 7.4.2 I/O

Write the quantum numbers in condensed form, enclosed by square brackets. For convenience, introduce also an array version.

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_write

<Quantum numbers: interfaces>+≡
    interface quantum_numbers_write
        module procedure quantum_numbers_write_single
        module procedure quantum_numbers_write_array
    end interface

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_write_single (qn, unit)
        type(quantum_numbers_t), intent(in) :: qn
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)", advance="no") "["
        if (flavor_is_defined (qn%f)) then
            call flavor_write (qn%f, u)
            if (color_is_defined (qn%c) .or. helicity_is_defined (qn%h)) &
                write (u, "(1x)", advance="no")
        end if
        if (color_is_defined (qn%c) .or. color_is_ghost (qn%c)) then
            call color_write (qn%c, u)
            if (helicity_is_defined (qn%h)) write (u, "(1x)", advance="no")
        end if
    end subroutine

```

```

        if (helicity_is_defined (qn%h)) then
            call helicity_write (qn%h, u)
        end if
        write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_write_single

    subroutine quantum_numbers_write_array (qn, unit)
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        integer, intent(in), optional :: unit
        integer :: i
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)", advance="no") "["
        do i = 1, size (qn)
            if (i > 1) write (u, "(A)", advance="no") " / "
            if (flavor_is_defined (qn(i)%f)) then
                call flavor_write (qn(i)%f, u)
                if (color_is_defined (qn(i)%c) .or. helicity_is_defined (qn(i)%h)) &
                    write (u, "(1x)", advance="no")
            end if
            if (color_is_defined (qn(i)%c) .or. color_is_ghost (qn(i)%c)) then
                call color_write (qn(i)%c, u)
                if (helicity_is_defined (qn(i)%h)) write (u, "(1x)", advance="no")
            end if
            if (helicity_is_defined (qn(i)%h)) then
                call helicity_write (qn(i)%h, u)
            end if
        end do
        write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_write_array

```

Binary I/O.

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_write_raw
    public :: quantum_numbers_read_raw

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_write_raw (qn, u)
        type(quantum_numbers_t), intent(in) :: qn
        integer, intent(in) :: u
        call flavor_write_raw (qn%f, u)
        call color_write_raw (qn%c, u)
        call helicity_write_raw (qn%h, u)
    end subroutine quantum_numbers_write_raw

    subroutine quantum_numbers_read_raw (qn, u, iostat)
        type(quantum_numbers_t), intent(out) :: qn
        integer, intent(in) :: u
        integer, intent(out), optional :: iostat
        call flavor_read_raw (qn%f, u, iostat=iostat)
        call color_read_raw (qn%c, u, iostat=iostat)
        call helicity_read_raw (qn%h, u, iostat=iostat)
    end subroutine quantum_numbers_read_raw

```

### 7.4.3 Accessing contents

Color and helicity can be done by elemental functions. Flavor needs explicit specifics because of the pointer assignment.

```
<Quantum numbers: public>+≡
    public :: quantum_numbers_get_flavor
    public :: quantum_numbers_get_color
    public :: quantum_numbers_get_helicity

<Quantum numbers: interfaces>+≡
    interface quantum_numbers_get_flavor
        module procedure quantum_numbers_get_flavor0
        module procedure quantum_numbers_get_flavor1
    end interface

<Quantum numbers: procedures>+≡
    function quantum_numbers_get_flavor0 (qn) result (flv)
        type(flavor_t) :: flv
        type(quantum_numbers_t), intent(in) :: qn
        flv = qn%f
    end function quantum_numbers_get_flavor0

    function quantum_numbers_get_flavor1 (qn) result (flv)
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        type(flavor_t), dimension(size(qn)) :: flv
        integer :: i
        do i = 1, size (qn)
            flv(i) = qn(i)%f
        end do
    end function quantum_numbers_get_flavor1

    elemental function quantum_numbers_get_color (qn) result (col)
        type(color_t) :: col
        type(quantum_numbers_t), intent(in) :: qn
        col = qn%c
    end function quantum_numbers_get_color

    elemental function quantum_numbers_get_helicity (qn) result (hel)
        type(helicity_t) :: hel
        type(quantum_numbers_t), intent(in) :: qn
        hel = qn%h
    end function quantum_numbers_get_helicity

<Quantum numbers: public>+≡
    public :: quantum_numbers_set_flavor
    public :: quantum_numbers_set_color
    public :: quantum_numbers_set_helicity

<Quantum numbers: interfaces>+≡
    interface quantum_numbers_set_flavor
        module procedure quantum_numbers_set_flavor0
        module procedure quantum_numbers_set_flavor1
    end interface
```

```

<Quantum numbers: procedures>+≡
  subroutine quantum_numbers_set_flavor0 (qn, flv)
    type(quantum_numbers_t), intent(inout) :: qn
    type(flavor_t), intent(in) :: flv
    qn%f = flv
  end subroutine quantum_numbers_set_flavor0

  subroutine quantum_numbers_set_flavor1 (qn, flv)
    type(quantum_numbers_t), dimension(:), intent(inout) :: qn
    type(flavor_t), dimension(:), intent(in) :: flv
    integer :: i
    do i = 1, size (flv)
      qn(i)%f = flv(i)
    end do
  end subroutine quantum_numbers_set_flavor1

  elemental subroutine quantum_numbers_set_color (qn, col)
    type(quantum_numbers_t), intent(inout) :: qn
    type(color_t), intent(in) :: col
    qn%c = col
  end subroutine quantum_numbers_set_color

  elemental subroutine quantum_numbers_set_helicity (qn, hel)
    type(quantum_numbers_t), intent(inout) :: qn
    type(helicity_t), intent(in) :: hel
    qn%h = hel
  end subroutine quantum_numbers_set_helicity

```

This just resets the ghost property of the color/helicity part:

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_set_color_ghost
  public :: quantum_numbers_set_helicity_ghost

<Quantum numbers: procedures>+≡
  elemental subroutine quantum_numbers_set_color_ghost (qn, ghost)
    type(quantum_numbers_t), intent(inout) :: qn
    logical, intent(in) :: ghost
    call color_set_ghost (qn%c, ghost)
  end subroutine quantum_numbers_set_color_ghost

  elemental subroutine quantum_numbers_set_helicity_ghost (qn, ghost)
    type(quantum_numbers_t), intent(inout) :: qn
    logical, intent(in) :: ghost
    call helicity_set_ghost (qn%h, ghost)
  end subroutine quantum_numbers_set_helicity_ghost

```

Assign a model to the flavor part of quantum numbers.

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_set_model

<Quantum numbers: interfaces>+≡
  interface quantum_numbers_set_model
    module procedure quantum_numbers_set_model_single
    module procedure quantum_numbers_set_model_array
  end interface

```



```

end interface

<Quantum numbers: procedures>+≡
  subroutine quantum_numbers_set_model_single (qn, model)
    type(quantum_numbers_t), intent(inout) :: qn
    type(model_t), intent(in), target :: model
    call flavor_set_model (qn%f, model)
  end subroutine quantum_numbers_set_model_single

  subroutine quantum_numbers_set_model_array (qn, model)
    type(quantum_numbers_t), dimension(:), intent(inout) :: qn
    type(model_t), intent(in), target :: model
    call flavor_set_model (qn%f, model)
  end subroutine quantum_numbers_set_model_array

```

This is a convenience function: return the color type for the flavor (array).

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_get_color_type

<Quantum numbers: procedures>+≡
  elemental function quantum_numbers_get_color_type (qn) result (color_type)
    integer :: color_type
    type(quantum_numbers_t), intent(in) :: qn
    color_type = flavor_get_color_type (qn%f)
  end function quantum_numbers_get_color_type

```

#### 7.4.4 Predicates

Check if the flavor index is valid (including UNDEFINED).

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_are_valid

<Quantum numbers: procedures>+≡
  elemental function quantum_numbers_are_valid (qn) result (valid)
    logical :: valid
    type(quantum_numbers_t), intent(in) :: qn
    valid = flavor_is_valid (qn%f)
  end function quantum_numbers_are_valid

```

Check if the flavor part has its particle-data pointer associated (debugging aid).

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_are_associated

<Quantum numbers: procedures>+≡
  elemental function quantum_numbers_are_associated (qn) result (flag)
    logical :: flag
    type(quantum_numbers_t), intent(in) :: qn
    flag = flavor_is_associated (qn%f)
  end function quantum_numbers_are_associated

```

Check if the helicity and color quantum numbers are diagonal. (Unpolarized/colorless also counts as diagonal.) Flavor is diagonal by definition.

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_are_diagonal

<Quantum numbers: procedures>+≡
  elemental function quantum_numbers_are_diagonal (qn) result (diagonal)
    logical :: diagonal
    type(quantum_numbers_t), intent(in) :: qn
    diagonal = helicity_is_diagonal (qn%h) .and. color_is_diagonal (qn%c)
  end function quantum_numbers_are_diagonal

```

Check if the color and/or helicity part has the ghost property.

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_is_color_ghost
  public :: quantum_numbers_is_helicity_ghost

<Quantum numbers: procedures>+≡
  elemental function quantum_numbers_is_color_ghost (qn) result (ghost)
    logical :: ghost
    type(quantum_numbers_t), intent(in) :: qn
    ghost = color_is_ghost (qn%c)
  end function quantum_numbers_is_color_ghost

  elemental function quantum_numbers_is_helicity_ghost (qn) result (ghost)
    logical :: ghost
    type(quantum_numbers_t), intent(in) :: qn
    ghost = helicity_is_ghost (qn%h)
  end function quantum_numbers_is_helicity_ghost

```

### 7.4.5 Comparisons

Matching and equality is derived from the individual quantum numbers. The variant `fhmatch` matches only flavor and helicity. The variant `dhmatch` matches only diagonal helicity, if the matching helicity is undefined.

```

<Quantum numbers: public>+≡
  public :: operator(.match.)
  public :: operator(.fmatch.)
  public :: operator(.fhmatch.)
  public :: operator(.dhmatch.)
  public :: operator(==)
  public :: operator(/=)

<Quantum numbers: interfaces>+≡
  interface operator(.match.)
    module procedure quantum_numbers_match
  end interface
  interface operator(.fmatch.)
    module procedure quantum_numbers_match_f
  end interface
  interface operator(.fhmatch.)
    module procedure quantum_numbers_match_fh
  end interface

```

```

interface operator(.dhmatch.)
  module procedure quantum_numbers_match_hel_diag
end interface
interface operator(==)
  module procedure quantum_numbers_eq
end interface
interface operator(/=)
  module procedure quantum_numbers_neq
end interface

(Quantum numbers: procedures) +=
  elemental function quantum_numbers_match (qn1, qn2) result (match)
    logical :: match
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    match = (qn1%f .match. qn2%f) .and. &
      (qn1%c .match. qn2%c) .and. &
      (qn1%h .match. qn2%h)
  end function quantum_numbers_match

  elemental function quantum_numbers_match_f (qn1, qn2) result (match)
    logical :: match
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    match = (qn1%f .match. qn2%f)
  end function quantum_numbers_match_f

  elemental function quantum_numbers_match_fh (qn1, qn2) result (match)
    logical :: match
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    match = (qn1%f .match. qn2%f) .and. &
      (qn1%h .match. qn2%h)
  end function quantum_numbers_match_fh

  elemental function quantum_numbers_match_hel_diag (qn1, qn2) result (match)
    logical :: match
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    match = (qn1%f .match. qn2%f) .and. &
      (qn1%c .match. qn2%c) .and. &
      (qn1%h .dhmatch. qn2%h)
  end function quantum_numbers_match_hel_diag

  elemental function quantum_numbers_eq (qn1, qn2) result (eq)
    logical :: eq
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    eq = (qn1%f == qn2%f) .and. &
      (qn1%c == qn2%c) .and. &
      (qn1%h == qn2%h)
  end function quantum_numbers_eq

  elemental function quantum_numbers_neq (qn1, qn2) result (neq)
    logical :: neq
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    neq = (qn1%f /= qn2%f) .or. &
      (qn1%c /= qn2%c) .or. &
      (qn1%h /= qn2%h)
  end function quantum_numbers_neq

```

Two sets of quantum numbers are compatible if the individual quantum numbers are compatible, depending on the mask. Flavor has to match, regardless of the flavor mask.

If the color flag is set, color is compatible if the ghost property is identical. If the color flag is unset, color has to be identical. I.e., if the flag is set, the color amplitudes can interfere. If it is not set, they must be identical, and there must be no ghost. The latter property is used for expanding physical color flows.

Helicity is compatible if the mask is unset, otherwise it has to match. This determines if two amplitudes can be multiplied (no mask) or traced (mask).

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_are_compatible

<Quantum numbers: procedures>+≡
    elemental function quantum_numbers_are_compatible (qn1, qn2, mask) &
        result (flag)
        logical :: flag
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        type(quantum_numbers_mask_t), intent(in) :: mask
        if (mask%h .or. mask%hd) then
            flag = (qn1%f .match. qn2%f) .and. (qn1%h .match. qn2%h)
        else
            flag = (qn1%f .match. qn2%f)
        end if
        if (mask%c) then
            flag = flag .and. (color_is_ghost (qn1%c) .eqv. color_is_ghost (qn2%c))
        else
            flag = flag .and. &
                .not. (color_is_ghost (qn1%c) .or. color_is_ghost (qn2%c)) .and. &
                (qn1%c == qn2%c)
        end if
    end function quantum_numbers_are_compatible

```

This is the analog for a single quantum-number set. We just check for color ghosts; they are excluded if the color mask is unset (color-flow expansion).

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_are_physical

<Quantum numbers: procedures>+≡
    elemental function quantum_numbers_are_physical (qn, mask) result (flag)
        logical :: flag
        type(quantum_numbers_t), intent(in) :: qn
        type(quantum_numbers_mask_t), intent(in) :: mask
        if (mask%c) then
            flag = .true.
        else
            flag = .not. color_is_ghost (qn%c)
        end if
    end function quantum_numbers_are_physical

```

## 7.4.6 Operations

Inherited from the color component: reassign color indices in canonical order.

```
<Quantum numbers: public>+≡
    public :: quantum_numbers_canonicalize_color

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_canonicalize_color (qn)
        type(quantum_numbers_t), dimension(:), intent(inout) :: qn
        call color_canonicalize (qn%c)
    end subroutine quantum_numbers_canonicalize_color
```

Inherited from the color component: make a color map for two matching quantum-number arrays.

```
<Quantum numbers: public>+≡
    public :: quantum_numbers_set_color_map

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_set_color_map (map, qn1, qn2)
        integer, dimension(:,:), intent(out), allocatable :: map
        type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
        call set_color_map (map, qn1%c, qn2%c)
    end subroutine quantum_numbers_set_color_map
```

Inherited from the color component: translate the color part using a color-map array

```
<Quantum numbers: public>+≡
    public :: quantum_numbers_translate_color

<Quantum numbers: interfaces>+≡
    interface quantum_numbers_translate_color
        module procedure quantum_numbers_translate_color0
        module procedure quantum_numbers_translate_color1
    end interface

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_translate_color0 (qn, map, offset)
        type(quantum_numbers_t), intent(inout) :: qn
        integer, dimension(:,:), intent(in) :: map
        integer, intent(in), optional :: offset
        call color_translate (qn%c, map, offset)
    end subroutine quantum_numbers_translate_color0

    subroutine quantum_numbers_translate_color1 (qn, map, offset)
        type(quantum_numbers_t), dimension(:), intent(inout) :: qn
        integer, dimension(:,:), intent(in) :: map
        integer, intent(in), optional :: offset
        call color_translate (qn%c, map, offset)
    end subroutine quantum_numbers_translate_color1
```

Inherited from the color component: return the color index with highest absolute value

```
<Quantum numbers: public>+≡
    public :: quantum_numbers_get_max_color_value
```

```

<Quantum numbers: interfaces>+≡
  interface quantum_numbers_get_max_color_value
    module procedure quantum_numbers_get_max_color_value0
    module procedure quantum_numbers_get_max_color_value1
    module procedure quantum_numbers_get_max_color_value2
  end interface

<Quantum numbers: procedures>+≡
  function quantum_numbers_get_max_color_value0 (qn) result (cmax)
    integer :: cmax
    type(quantum_numbers_t), intent(in) :: qn
    cmax = color_get_max_value0 (qn%c)
    !!! ifort 11.1 v5 demands this
    !!! cmax = color_get_max_value (qn%c)
  end function quantum_numbers_get_max_color_value0

  function quantum_numbers_get_max_color_value1 (qn) result (cmax)
    integer :: cmax
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    cmax = color_get_max_value1 (qn%c)
    !!! ifort 11.1 v5 demands this
    !!! cmax = color_get_max_value (qn%c)
  end function quantum_numbers_get_max_color_value1

  function quantum_numbers_get_max_color_value2 (qn) result (cmax)
    integer :: cmax
    type(quantum_numbers_t), dimension(:,:), intent(in) :: qn
    cmax = color_get_max_value2 (qn%c)
    !!! ifort 11.1 v5 demands this
    !!! cmax = color_get_max_value (qn%c)
  end function quantum_numbers_get_max_color_value2

```

Inherited from the color component: add an offset to the indices of the color part

```

<Quantum numbers: public>+≡
  public :: quantum_numbers_add_color_offset

<Quantum numbers: procedures>+≡
  elemental subroutine quantum_numbers_add_color_offset (qn, offset)
    type(quantum_numbers_t), intent(inout) :: qn
    integer, intent(in) :: offset
    call color_add_offset (qn%c, offset)
  end subroutine quantum_numbers_add_color_offset

```

Given a quantum number array, return all possible color contractions, leaving the other quantum numbers intact.

```

<Quantum numbers: public>+≡
  public :: quantum_number_array_make_color_contractions

<Quantum numbers: procedures>+≡
  subroutine quantum_number_array_make_color_contractions (qn_in, qn_out)
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    type(quantum_numbers_t), dimension(:,:), intent(out), allocatable :: qn_out

```

```

type(color_t), dimension(:,:), allocatable :: col
integer :: i
call color_array_make_contractions (qn_in%c, col)
allocate (qn_out (size (col, 1), size (col, 2)))
do i = 1, size (qn_out, 2)
    qn_out(:,i)%f = qn_in%f
    qn_out(:,i)%c = col(:,i)
    qn_out(:,i)%h = qn_in%h
end do
end subroutine quantum_number_array_make_color_contractions

```

Inherited from the color component: invert the color, switching particle/antiparticle.

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_invert_color

<Quantum numbers: procedures>+≡
    elemental subroutine quantum_numbers_invert_color (qn)
        type(quantum_numbers_t), intent(inout) :: qn
        call color_invert (qn%c)
    end subroutine quantum_numbers_invert_color

```

Merge two quantum number sets: for each entry, if both are defined, combine them to an off-diagonal entry (meaningful only if the input was diagonal). If either entry is undefined, take the defined one.

For flavor, off-diagonal entries are invalid, so both flavors must be equal, otherwise an invalid flavor is inserted.

```

<Quantum numbers: public>+≡
    public :: operator(.merge.)

<Quantum numbers: interfaces>+≡
    interface operator(.merge.)
        module procedure merge_quantum_numbers0
        module procedure merge_quantum_numbers1
    end interface

<Quantum numbers: procedures>+≡
    function merge_quantum_numbers0 (qn1, qn2) result (qn3)
        type(quantum_numbers_t) :: qn3
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        qn3%f = qn1%f .merge. qn2%f
        qn3%c = qn1%c .merge. qn2%c
        qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers0

    function merge_quantum_numbers1 (qn1, qn2) result (qn3)
        type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
        type(quantum_numbers_t), dimension(size(qn1)) :: qn3
        qn3%f = qn1%f .merge. qn2%f
        qn3%c = qn1%c .merge. qn2%c
        qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers1

```

### 7.4.7 The quantum number mask

The quantum numbers mask is true for quantum numbers that should be ignored or summed over. The three mandatory entries correspond to flavor, color, and helicity, respectively.

There is an additional entry `cg`: If false, the color-ghosts property should be kept even if color is ignored. This is relevant only if `c` is set, otherwise it is always false.

The flag `hd` tells that only diagonal entries in helicity should be kept. If `h` is set, `hd` is irrelevant and will be kept `.false.`

```
<Quantum numbers: public>+≡
  public :: quantum_numbers_mask_t

<Quantum numbers: types>+≡
  type :: quantum_numbers_mask_t
    private
    logical :: f = .false.
    logical :: c = .false.
    logical :: cg = .false.
    logical :: h = .false.
    logical :: hd = .false.
  end type quantum_numbers_mask_t
```

Define a quantum number mask: Constructor form

```
<Quantum numbers: public>+≡
  public :: new_quantum_numbers_mask

<Quantum numbers: procedures>+≡
  elemental function new_quantum_numbers_mask &
    (mask_f, mask_c, mask_h, mask_cg, mask_hd) result (mask)
    type(quantum_numbers_mask_t) :: mask
    logical, intent(in) :: mask_f, mask_c, mask_h
    logical, intent(in), optional :: mask_cg
    logical, intent(in), optional :: mask_hd
    call quantum_numbers_mask_init &
      (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
  end function new_quantum_numbers_mask
```

Define quantum numbers: Initializer form

```
<Quantum numbers: public>+≡
  public :: quantum_numbers_mask_init

<Quantum numbers: procedures>+≡
  elemental subroutine quantum_numbers_mask_init &
    (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
    type(quantum_numbers_mask_t), intent(out) :: mask
    logical, intent(in) :: mask_f, mask_c, mask_h
    logical, intent(in), optional :: mask_cg, mask_hd
    mask%f = mask_f
    mask%c = mask_c
    mask%h = mask_h
    if (present (mask_cg)) then
      if (mask%c) mask%cg = mask_cg
    else
```



```

        mask%cg = mask_c
    end if
    if (present (mask_hd)) then
        if (.not. mask%h) mask%hd = mask_hd
    end if
end subroutine quantum_numbers_mask_init

```

Write a quantum numbers mask

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_mask_write

<Quantum numbers: interfaces>+≡
    interface quantum_numbers_mask_write
        module procedure quantum_numbers_mask_write_single
        module procedure quantum_numbers_mask_write_array
    end interface

<Quantum numbers: procedures>+≡
    subroutine quantum_numbers_mask_write_single (mask, unit)
        type(quantum_numbers_mask_t), intent(in) :: mask
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)", advance="no") "["
        write (u, "(L1)", advance="no") mask%f
        write (u, "(L1)", advance="no") mask%c
        if (.not.mask%cg) write (u, "('g')", advance="no")
        write (u, "(L1)", advance="no") mask%h
        if (mask%hd) write (u, "('d')", advance="no")
        write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_single

    subroutine quantum_numbers_mask_write_array (mask, unit)
        type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit); if (u < 0) return
        write (u, "(A)", advance="no") "["
        do i = 1, size (mask)
            if (i > 1) write (u, "(A)", advance="no") "/"
            write (u, "(L1)", advance="no") mask(i)%f
            write (u, "(L1)", advance="no") mask(i)%c
            if (.not.mask(i)%cg) write (u, "('g')", advance="no")
            write (u, "(L1)", advance="no") mask(i)%h
            if (mask(i)%hd) write (u, "('d')", advance="no")
        end do
        write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_array

```

## 7.4.8 Setting mask components

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_mask_set_flavor

```

```

public :: quantum_numbers_mask_set_color
public :: quantum_numbers_mask_set_helicity

<Quantum numbers: procedures>+≡
elemental subroutine quantum_numbers_mask_set_flavor (mask, mask_f)
  type(quantum_numbers_mask_t), intent(inout) :: mask
  logical, intent(in) :: mask_f
  mask%f = mask_f
end subroutine quantum_numbers_mask_set_flavor

elemental subroutine quantum_numbers_mask_set_color (mask, mask_c, mask_cg)
  type(quantum_numbers_mask_t), intent(inout) :: mask
  logical, intent(in) :: mask_c
  logical, intent(in), optional :: mask_cg
  mask%c = mask_c
  if (present (mask_cg)) then
    if (mask%c) mask%cg = mask_cg
  else
    mask%cg = mask_c
  end if
end subroutine quantum_numbers_mask_set_color

elemental subroutine quantum_numbers_mask_set_helicity (mask, mask_h, mask_hd)
  type(quantum_numbers_mask_t), intent(inout) :: mask
  logical, intent(in) :: mask_h
  logical, intent(in), optional :: mask_hd
  mask%h = mask_h
  if (present (mask_hd)) then
    if (.not. mask%h) mask%hd = mask_hd
  end if
end subroutine quantum_numbers_mask_set_helicity

```

The following routines assign part of a mask, depending on the flags given.

```

<Quantum numbers: public>+≡
public :: quantum_numbers_mask_assign

<Quantum numbers: procedures>+≡
elemental subroutine quantum_numbers_mask_assign &
  (mask, mask_in, flavor, color, helicity)
  type(quantum_numbers_mask_t), intent(inout) :: mask
  type(quantum_numbers_mask_t), intent(in) :: mask_in
  logical, intent(in), optional :: flavor, color, helicity
  if (present (flavor)) then
    if (flavor) then
      mask%f = mask_in%f
    end if
  end if
  if (present (color)) then
    if (color) then
      mask%c = mask_in%c
      mask%cg = mask_in%cg
    end if
  end if
  if (present (helicity)) then
    if (helicity) then

```

```

        mask%h = mask_in%h
        mask%hd = mask_in%hd
    end if
end if
end subroutine quantum_numbers_mask_assign

```

#### 7.4.9 Mask predicates

Return true if either one of the entries is set:

```

<Quantum numbers: public>+≡
    public :: any

<Quantum numbers: interfaces>+≡
    interface any
        module procedure quantum_numbers_mask_any
    end interface

<Quantum numbers: procedures>+≡
    function quantum_numbers_mask_any (mask) result (match)
        logical :: match
        type(quantum_numbers_mask_t), intent(in) :: mask
        match = mask%f .or. mask%c .or. mask%h .or. mask%hd
    end function quantum_numbers_mask_any

```

#### 7.4.10 Operators

The OR operation is applied to all components.

```

<Quantum numbers: public>+≡
    public :: operator(.or.)

<Quantum numbers: interfaces>+≡
    interface operator(.or.)
        module procedure quantum_numbers_mask_or
    end interface

<Quantum numbers: procedures>+≡
    elemental function quantum_numbers_mask_or (mask1, mask2) result (mask)
        type(quantum_numbers_mask_t) :: mask
        type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
        mask%f = mask1%f .or. mask2%f
        mask%c = mask1%c .or. mask2%c
        if (mask%c) mask%cg = mask1%cg .or. mask2%cg
        mask%h = mask1%h .or. mask2%h
        if (.not. mask%h) mask%hd = mask1%hd .or. mask2%hd
    end function quantum_numbers_mask_or

```

#### 7.4.11 Mask comparisons

Return true if the two masks are equivalent / differ:

```

<Quantum numbers: public>+≡

```

```

public :: operator(.eqv.)
public :: operator(.neqv.)

<Quantum numbers: interfaces>+≡
interface operator(.eqv.)
  module procedure quantum_numbers_mask_eqv
end interface
interface operator(.neqv.)
  module procedure quantum_numbers_mask_neqv
end interface

<Quantum numbers: procedures>+≡
elemental function quantum_numbers_mask_eqv (mask1, mask2) result (eqv)
  logical :: eqv
  type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  eqv = (mask1%f .eqv. mask2%f) .and. &
        (mask1%c .eqv. mask2%c) .and. &
        (mask1%cg .eqv. mask2%cg) .and. &
        (mask1%h .eqv. mask2%h) .and. &
        (mask1%hd .eqv. mask2%hd)
end function quantum_numbers_mask_eqv

elemental function quantum_numbers_mask_neqv (mask1, mask2) result (neqv)
  logical :: neqv
  type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  neqv = (mask1%f .neqv. mask2%f) .or. &
        (mask1%c .neqv. mask2%c) .or. &
        (mask1%cg .neqv. mask2%cg) .or. &
        (mask1%h .neqv. mask2%h) .or. &
        (mask1%hd .neqv. mask2%hd)
end function quantum_numbers_mask_neqv

```

#### 7.4.12 Apply a mask

Applying a mask to the quantum number object means undefining those entries where the mask is set. The others remain unaffected.

The `hd` mask has the special property that it “diagonalizes” helicity, i.e., the second helicity entry is dropped and the result is a diagonal helicity quantum number.

```

<Quantum numbers: public>+≡
public :: quantum_numbers_undefine
public :: quantum_numbers_undefined

<Quantum numbers: interfaces>+≡
interface quantum_numbers_undefined
  module procedure quantum_numbers_undefined0
  module procedure quantum_numbers_undefined1
  module procedure quantum_numbers_undefined11
end interface

<Quantum numbers: procedures>+≡
elemental subroutine quantum_numbers_undefine (qn, mask)
  type(quantum_numbers_t), intent(inout) :: qn

```

```

type(quantum_numbers_mask_t), intent(in) :: mask
if (mask%f) call flavor_undefine (qn%f)
if (mask%c) call color_undefine (qn%c, undefine_ghost=mask%cg)
if (mask%h) then
    call helicity_undefine (qn%h)
else if (mask%hd) then
    if (.not. helicity_is_diagonal (qn%h)) then
        call helicity_diagonalize (qn%h)
    end if
end if
end if
end subroutine quantum_numbers_undefine

function quantum_numbers_undefined0 (qn, mask) result (qn_new)
type(quantum_numbers_t), intent(in) :: qn
type(quantum_numbers_mask_t), intent(in) :: mask
type(quantum_numbers_t) :: qn_new
qn_new = qn
call quantum_numbers_undefine (qn_new, mask)
end function quantum_numbers_undefined0

function quantum_numbers_undefined1 (qn, mask) result (qn_new)
type(quantum_numbers_t), dimension(:), intent(in) :: qn
type(quantum_numbers_mask_t), intent(in) :: mask
type(quantum_numbers_t), dimension(size(qn)) :: qn_new
qn_new = qn
call quantum_numbers_undefine (qn_new, mask)
end function quantum_numbers_undefined1

function quantum_numbers_undefined11 (qn, mask) result (qn_new)
type(quantum_numbers_t), dimension(:), intent(in) :: qn
type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
type(quantum_numbers_t), dimension(size(qn)) :: qn_new
qn_new = qn
call quantum_numbers_undefine (qn_new, mask)
end function quantum_numbers_undefined11

```

Return true if the input quantum number set has entries that would be removed by the applied mask, e.g., if polarization is defined but `mask%h` is set:

```

<Quantum numbers: public>+≡
public :: quantum_numbers_are_redundant

<Quantum numbers: procedures>+≡
elemental function quantum_numbers_are_redundant (qn, mask) &
    result (redundant)
logical :: redundant
type(quantum_numbers_t), intent(in) :: qn
type(quantum_numbers_mask_t), intent(in) :: mask
redundant = .false.
if (mask%f) then
    redundant = flavor_is_defined (qn%f)
end if
if (mask%c) then
    redundant = color_is_defined (qn%c)
end if

```

```

    if (mask%h) then
        redundant = helicity_is_defined (qn%h)
    else if (mask%hd) then
        redundant = .not. helicity_is_diagonal (qn%h)
    end if
end function quantum_numbers_are_redundant

```

Return true if the helicity flag is set or the diagonal-helicity flag is set.

```

<Quantum numbers: public>+≡
    public :: quantum_numbers_mask_diagonal_helicity

<Quantum numbers: procedures>+≡
    elemental function quantum_numbers_mask_diagonal_helicity (mask) &
        result (flag)
        logical :: flag
        type(quantum_numbers_mask_t), intent(in) :: mask
        flag = mask%h .or. mask%hd
    end function quantum_numbers_mask_diagonal_helicity

```

## 7.5 State matrices

This module deals with the internal state of a particle system, i.e., with its density matrix in flavor, color, and helicity space.

```
<state_matrices.f90>≡  
<File header>  
  
module state_matrices  
  
  <Use kinds>  
  <Use file utils>  
  use diagnostics !NODEP!  
  use models  
  use flavors  
  use colors  
  use helicities  
  use quantum_numbers  
  
  <Standard module head>  
  
  <State matrices: public>  
  
  <State matrices: parameters>  
  
  <State matrices: types>  
  
  <State matrices: interfaces>  
  
contains  
  
  <State matrices: procedures>  
  
end module state_matrices
```

### 7.5.1 Nodes of the quantum state trie

A quantum state object represents an unnormalized density matrix, i.e., an array of possibilities for flavor, color, and helicity indices with associated complex values. Physically, the trace of this matrix is the summed squared matrix element for an interaction, and the matrix elements divided by this value correspond to the flavor-color-helicity density matrix. (Flavor and color are diagonal.)

We store density matrices as tries, that is, as trees where each branching represents the possible quantum numbers of a particle. The first branching is the first particle in the system. A leaf (the node corresponding to the last particle) contains the value of the matrix element.

Each node contains a flavor, color, and helicity entry. Note that each of those entries may be actually undefined, so we can also represent, e.g., unpolarized particles.

The value is meaningful only for leaves, which have no child nodes. There is a pointer to the parent node which allows for following the trie downwards from a leaf, it is null for a root node. The child nodes are implemented as a list,

so there is a pointer to the first and last child, and each node also has a **next** pointer to the next sibling.

The root node does not correspond to a particle, only its children do. The quantum numbers of the root node are irrelevant and will not be set. However, we use a common type for the three classes (root, branch, leaf); they may easily be distinguished by the association status of parent and child.

## Node type

The node is linked in all directions: the parent, the first and last in the list of children, and the previous and next sibling. This allows us for adding and removing nodes and whole branches anywhere in the trie. (Circular links are not allowed, however.). The node holds its associated set of quantum numbers. The integer index, which is set only for leaf nodes, is the index of the corresponding matrix element value within the state matrix.

Temporarily, matrix-element values may be stored within a leaf node. This is used during state-matrix factorization. When the state matrix is **frozen**, these values are transferred to the matrix-element array within the host state matrix.

```

<State matrices: types>≡
  type :: node_t
    private
    type(quantum_numbers_t) :: qn
    type(node_t), pointer :: parent => null ()
    type(node_t), pointer :: child_first => null ()
    type(node_t), pointer :: child_last => null ()
    type(node_t), pointer :: next => null ()
    type(node_t), pointer :: previous => null ()
    integer :: me_index = 0
    integer, dimension(:), allocatable :: me_count
    complex(default) :: me = 0
  end type node_t

```

## Operations on nodes

Recursively deallocate all children of the current node. This includes any values associated with the children.

```

<State matrices: procedures>≡
  pure recursive subroutine node_delete_offspring (node)
    type(node_t), pointer :: node
    type(node_t), pointer :: child
    child => node%child_first
    do while (associated (child))
      node%child_first => node%child_first%next
      call node_delete_offspring (child)
      deallocate (child)
    end do
    node%child_last => null ()
  end subroutine node_delete_offspring

```



Remove a node including its offspring. Adjust the pointers of parent and siblings, if necessary.

```

<State matrices: procedures>+≡
  pure subroutine node_delete (node)
    type(node_t), pointer :: node
    call node_delete_offspring (node)
    if (associated (node%previous)) then
      node%previous%next => node%next
    else if (associated (node%parent)) then
      node%parent%child_first => node%next
    end if
    if (associated (node%next)) then
      node%next%previous => node%previous
    else if (associated (node%parent)) then
      node%parent%child_last => node%previous
    end if
    deallocate (node)
  end subroutine node_delete

```

Append a child node

```

<State matrices: procedures>+≡
  subroutine node_append_child (node, child)
    type(node_t), target, intent(inout) :: node
    type(node_t), pointer :: child
    allocate (child)
    if (associated (node%child_last)) then
      node%child_last%next => child
      child%previous => node%child_last
    else
      node%child_first => child
    end if
    node%child_last => child
    child%parent => node
  end subroutine node_append_child

```

## I/O

Output of a single node, no recursion. We print the quantum numbers in square brackets, then the value (if any).

```

<State matrices: procedures>+≡
  subroutine node_write (node, me_array, verbose, unit)
    type(node_t), intent(in) :: node
    complex(default), dimension(:), intent(in), optional :: me_array
    logical, intent(in), optional :: verbose
    integer, intent(in), optional :: unit
    logical :: verb
    integer :: u
    verb = .false.; if (present (verbose)) verb = verbose
    u = output_unit (unit); if (u < 0) return
    call quantum_numbers_write (node%qn, u)
    if (node%me_index /= 0) then
      write (u, "(A,I0,A)", advance="no") " => ME(", node%me_index, ")"

```

```

        if (present (me_array)) then
            write (u, "(A)", advance="no") " = "
            write (u, "('(',ES19.12,',',',ES19.12,')')'", advance="no") &
                me_array(node%me_index)
        end if
    end if
write (u, *)
if (verb) then
    call ptr_write ("parent      ", node%parent)
    call ptr_write ("child_first", node%child_first)
    call ptr_write ("child_last ", node%child_last)
    call ptr_write ("next        ", node%next)
    call ptr_write ("previous   ", node%previous)
end if
contains
subroutine ptr_write (label, node)
    character(*), intent(in) :: label
    type(node_t), pointer :: node
    if (associated (node)) then
        write (u, "(10x,A,1x,'->',1x)", advance="no") label
        call quantum_numbers_write (node%qn, u)
        write (u, *)
    end if
end subroutine ptr_write
end subroutine node_write

```

Recursive output of a node:

```

<State matrices: procedures>+≡
recursive subroutine node_write_rec (node, me_array, verbose, indent, unit)
    type(node_t), intent(in), target :: node
    complex(default), dimension(:), intent(in), optional :: me_array
    logical, intent(in), optional :: verbose
    integer, intent(in), optional :: indent
    integer, intent(in), optional :: unit
    type(node_t), pointer :: current
    logical :: verb
    integer :: i, u
    verb = .false.; if (present (verbose)) verb = verbose
    i = 0; if (present (indent)) i = indent
    u = output_unit (unit); if (u < 0) return
    current => node%child_first
    do while (associated (current))
        write (u, "(A)", advance="no") repeat (" ", i)
        call node_write (current, me_array, verb, u)
        call node_write_rec (current, me_array, verb, i+2, u)
        current => current%next
    end do
end subroutine node_write_rec

```

Binary I/O. Matrix elements are written only for leaf nodes. this was actually a pointer is lost.

```

<State matrices: procedures>+≡
recursive subroutine node_write_raw_rec (node, u)

```

```

type(node_t), intent(in), target :: node
integer, intent(in) :: u
logical :: associated_child_first, associated_next
call quantum_numbers_write_raw (node%qn, u)
associated_child_first = associated (node%child_first)
write (u) associated_child_first
associated_next = associated (node%next)
write (u) associated_next
if (associated_child_first) then
    call node_write_raw_rec (node%child_first, u)
else
    write (u) node%me_index
end if
if (associated_next) then
    call node_write_raw_rec (node%next, u)
end if
end subroutine node_write_raw_rec

recursive subroutine node_read_raw_rec (node, u, parent, iostat)
type(node_t), intent(out), target :: node
integer, intent(in) :: u
type(node_t), intent(in), optional, target :: parent
integer, intent(out), optional :: iostat
logical :: associated_child_first, associated_next
type(node_t), pointer :: child
call quantum_numbers_read_raw (node%qn, u, iostat=iostat)
read (u, iostat=iostat) associated_child_first
read (u, iostat=iostat) associated_next
if (present (parent)) node%parent => parent
if (associated_child_first) then
    allocate (child)
    node%child_first => child
    node%child_last => null ()
    call node_read_raw_rec (child, u, node, iostat=iostat)
    do while (associated (child))
        child%previous => node%child_last
        node%child_last => child
        child => child%next
    end do
else
    read (u, iostat=iostat) node%me_index
end if
if (associated_next) then
    allocate (node%next)
    call node_read_raw_rec (node%next, u, parent, iostat=iostat)
end if
end subroutine node_read_raw_rec

```

## 7.5.2 State matrix

### Definition

The quantum state object is a container that keeps and hides the root node. For direct accessibility of values, they are stored in a separate array. The leaf nodes of the quantum-number tree point to those values, once the state matrix is finalized.

The `norm` component is redefined if a common factor is extracted from all nodes.

```

<State matrices: public>≡
  public :: state_matrix_t

<State matrices: types>+≡
  type :: state_matrix_t
    private
      type(node_t), pointer :: root => null ()
      integer :: depth = 0
      integer :: n_matrix_elements = 0
      logical :: leaf_nodes_store_values = .false.
      integer :: n_counters = 0
      complex(default), dimension(:), allocatable :: me
      real(default) :: norm = 1
  end type state_matrix_t

```

This initializer allocates the root node but does not fill anything. We declare whether values are stored within the nodes during state-matrix construction, and how many counters should be maintained (default: none).

```

<State matrices: public>+≡
  public :: state_matrix_init

<State matrices: procedures>+≡
  elemental subroutine state_matrix_init (state, store_values, n_counters)
    type(state_matrix_t), intent(out) :: state
    logical, intent(in), optional :: store_values
    integer, intent(in), optional :: n_counters
    allocate (state%root)
    if (present (store_values)) &
      state%leaf_nodes_store_values = store_values
    if (present (n_counters)) state%n_counters = n_counters
  end subroutine state_matrix_init

```

This recursively deletes all children of the root node, restoring the initial state. The matrix element array is not finalized, since it does not contain physical entries, just pointers.

```

<State matrices: public>+≡
  public :: state_matrix_final

<State matrices: procedures>+≡
  elemental subroutine state_matrix_final (state)
    type(state_matrix_t), intent(inout) :: state
    if (allocated (state%me)) deallocate (state%me)
    if (associated (state%root)) call node_delete (state%root)
    state%depth = 0
  end subroutine state_matrix_final

```

```

state%n_matrix_elements = 0
end subroutine state_matrix_final

```

Output: Present the tree as a nested list with appropriate indentation.

```

<State matrices: public>+≡
public :: state_matrix_write

<State matrices: procedures>+≡
subroutine state_matrix_write (state, unit, write_value_list, verbose)
  type(state_matrix_t), intent(in) :: state
  logical, intent(in), optional :: write_value_list, verbose
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i
  u = output_unit (unit); if (u < 0) return
  write (u, "(1x,A,ES19.12)") "State matrix: norm = ", state%norm
  if (associated (state%root)) then
    if (allocated (state%me)) then
      call node_write_rec (state%root, state%me, verbose, 1, u)
    else
      call node_write_rec (state%root, verbose=verbose, indent=1, unit=u)
    end if
  end if
  if (present (write_value_list)) then
    if (write_value_list .and. allocated (state%me)) then
      do i = 1, size (state%me)
        write (u, "(1x,I0,A)", advance="no") i, ":"
        write (u, "('(',ES19.12,',',',ES19.12,')')") state%me(i)
      end do
    end if
  end if
end subroutine state_matrix_write

```

Binary I/O. The auxiliary matrix-element array is not written, but reconstructed after reading the tree.

```

<State matrices: public>+≡
public :: state_matrix_write_raw
public :: state_matrix_read_raw

<State matrices: procedures>+≡
subroutine state_matrix_write_raw (state, u)
  type(state_matrix_t), intent(in) :: state
  integer, intent(in) :: u
  logical :: associated_root
  associated_root = associated (state%root)
  write (u) associated_root
  if (associated_root) then
    write (u) state%depth
    write (u) state%norm
    call node_write_raw_rec (state%root, u)
  end if
end subroutine state_matrix_write_raw

subroutine state_matrix_read_raw (state, u, iostat)

```

```

type(state_matrix_t), intent(out) :: state
integer, intent(in) :: u
integer, intent(out), optional :: iostat
logical :: associated_root
read (u, iostat=iostat) associated_root
if (associated_root) then
    read (u, iostat=iostat) state%depth
    read (u, iostat=iostat) state%norm
    call state_matrix_init (state)
    call node_read_raw_rec (state%root, u, iostat=iostat)
    call state_matrix_freeze (state)
end if
end subroutine state_matrix_read_raw

```

## Properties of the quantum state

A state is defined if its root is allocated:

```

<State matrices: public>+≡
    public :: state_matrix_is_defined

<State matrices: procedures>+≡
    elemental function state_matrix_is_defined (state) result (defined)
        logical :: defined
        type(state_matrix_t), intent(in) :: state
        defined = associated (state%root)
    end function state_matrix_is_defined

```

A state is empty if its depth is zero:

```

<State matrices: public>+≡
    public :: state_matrix_is_empty

<State matrices: procedures>+≡
    elemental function state_matrix_is_empty (state) result (flag)
        logical :: flag
        type(state_matrix_t), intent(in) :: state
        flag = state%depth == 0
    end function state_matrix_is_empty

```

Return the number of matrix-element values.

```

<State matrices: public>+≡
    public :: state_matrix_get_n_matrix_elements

<State matrices: procedures>+≡
    function state_matrix_get_n_matrix_elements (state) result (n)
        integer :: n
        type(state_matrix_t), intent(in) :: state
        n = state%n_matrix_elements
    end function state_matrix_get_n_matrix_elements

```

Return the number of leaves. This can be larger than the number of independent matrix elements.

```

<State matrices: public>+≡
    public :: state_matrix_get_n_leaves

```

```

<State matrices: procedures>+≡
function state_matrix_get_n_leaves (state) result (n)
  integer :: n
  type(state_matrix_t), intent(in) :: state
  type(state_iterator_t) :: it
  n = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
    n = n + 1
    call state_iterator_advance (it)
  end do
end function state_matrix_get_n_leaves

```

Return the depth:

```

<State matrices: public>+≡
public :: state_matrix_get_depth

<State matrices: procedures>+≡
function state_matrix_get_depth (state) result (depth)
  integer :: depth
  type(state_matrix_t), intent(in) :: state
  depth = state%depth
end function state_matrix_get_depth

```

Return the norm:

```

<State matrices: public>+≡
public :: state_matrix_get_norm

<State matrices: procedures>+≡
function state_matrix_get_norm (state) result (norm)
  real(default) :: norm
  type(state_matrix_t), intent(in) :: state
  norm = state%norm
end function state_matrix_get_norm

```

## Retrieving contents

Return the quantum number array, using an index. We have to scan the state matrix since there is no shortcut.

```

<State matrices: public>+≡
public :: state_matrix_get_quantum_numbers

<State matrices: procedures>+≡
function state_matrix_get_quantum_numbers (state, i) result (qn)
  type(state_matrix_t), intent(in), target :: state
  integer, intent(in) :: i
  type(quantum_numbers_t), dimension(state%depth) :: qn
  type(state_iterator_t) :: it
  integer :: k
  k = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))

```

```

        k = k + 1
        if (k == i) then
            qn = state_iterator_get_quantum_numbers (it)
            return
        end if
        call state_iterator_advance (it)
    end do
end function state_matrix_get_quantum_numbers

```

Return a single matrix element using its index. Works only if the shortcut array is allocated.

```

<State matrices: public>+≡
    public :: state_matrix_get_matrix_element

<State matrices: procedures>+≡
    function state_matrix_get_matrix_element (state, i) result (me)
        complex(default) :: me
        type(state_matrix_t), intent(in) :: state
        integer, intent(in) :: i
        if (associated (state%me)) then
            me = state%me(i)
        else
            me = 0
        end if
    end function state_matrix_get_matrix_element

```

Return the color index with maximum absolute value that is present within the state matrix.

```

<State matrices: public>+≡
    public :: state_matrix_get_max_color_value

<State matrices: procedures>+≡
    function state_matrix_get_max_color_value (state) result (cmax)
        integer :: cmax
        type(state_matrix_t), intent(in) :: state
        if (associated (state%root)) then
            cmax = node_get_max_color_value (state%root)
        else
            cmax = 0
        end if
    contains
        recursive function node_get_max_color_value (node) result (cmax)
            integer :: cmax
            type(node_t), intent(in), target :: node
            type(node_t), pointer :: current
            cmax = quantum_numbers_get_max_color_value (node%qn)
            current => node%child_first
            do while (associated (current))
                cmax = max (cmax, node_get_max_color_value (current))
                current => current%next
            end do
        end function node_get_max_color_value
    end function state_matrix_get_max_color_value

```



## Building the quantum state

The procedure generates a branch associated to the input array of quantum numbers. If the branch exists already, it is used.

Optionally, we set the matrix-element index, a value (which may be added to the previous one), and increment one of the possible counters. We may also return the matrix element index of the current node.

```
(State matrices: public)+≡
    public :: state_matrix_add_state

(State matrices: procedures)+≡
    subroutine state_matrix_add_state &
        (state, qn, index, value, sum_values, counter_index, me_index)
        type(state_matrix_t), intent(inout) :: state
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        integer, intent(in), optional :: index
        complex(default), intent(in), optional :: value
        logical, intent(in), optional :: sum_values
        integer, intent(in), optional :: counter_index
        integer, intent(out), optional :: me_index
        logical :: set_index, get_index, add
        set_index = present (index)
        get_index = present (me_index)
        add = .false.; if (present (sum_values)) add = sum_values
        if (state%depth == 0) then
            state%depth = size (qn)
        else if (state%depth /= size (qn)) then
            call state_matrix_write (state)
            call msg_bug ("State matrix: depth mismatch")
        end if
        if (size (qn) > 0) call node_make_branch (state%root, qn)
contains
        recursive subroutine node_make_branch (parent, qn)
            type(node_t), pointer :: parent
            type(quantum_numbers_t), dimension(:), intent(in) :: qn
            type(node_t), pointer :: child
            logical :: match
            match = .false.
            child => parent%child_first
            SCAN_CHILDREN: do while (associated (child))
                match = child%qn == qn(1)
                if (match) exit SCAN_CHILDREN
                child => child%next
            end do SCAN_CHILDREN
            if (.not. match) then
                call node_append_child (parent, child)
                child%qn = qn(1)
            end if
            select case (size (qn))
            case (1)
                if (.not. match) then
                    state%n_matrix_elements = state%n_matrix_elements + 1
                    child%me_index = state%n_matrix_elements
                end if
```

```

    if (set_index) then
        child%me_index = index
    end if
    if (get_index) then
        me_index = child%me_index
    end if
    if (present (counter_index)) then
        if (.not. allocated (child%me_count)) then
            allocate (child%me_count (state%n_counters))
            child%me_count = 0
        end if
        child%me_count(counter_index) = child%me_count(counter_index) + 1
    end if
    if (present (value)) then
        if (add) then
            child%me = child%me + value
        else
            child%me = value
        end if
    end if
    case (2:)
        call node_make_branch (child, qn(2:))
    end select
end subroutine node_make_branch
end subroutine state_matrix_add_state

```

Remove irrelevant flavor/color/helicity labels and the corresponding branchings. The masks indicate which particles are affected; the masks length should coincide with the depth of the trie (without the root node). Recursively scan the whole tree, starting from the leaf nodes and working up to the root node. If a mask entry is set for the current tree level, scan the children there. For each child within that level make a new empty branch where the masked quantum number is undefined. Then recursively combine all following children with matching quantum number into this new node and move on.

```

<State matrices: public>+≡
    public :: state_matrix_collapse

<State matrices: procedures>+≡
    subroutine state_matrix_collapse (state, mask)
        type(state_matrix_t), intent(inout) :: state
        type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
        type(state_matrix_t) :: red_state
        if (state_matrix_is_defined (state)) then
            call state_matrix_reduce (state, mask, red_state)
            call state_matrix_final (state)
            state = red_state
        end if
    end subroutine state_matrix_collapse

```

Transform the given state matrix into a reduced state matrix where some quantum numbers are removed, as indicated by the mask. The procedure creates a new state matrix, so the old one can be deleted after this if it is no longer used.

```

<State matrices: public>+≡

```

```

public :: state_matrix_reduce
<State matrices: procedures>+≡
subroutine state_matrix_reduce (state, mask, red_state)
  type(state_matrix_t), intent(in), target :: state
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
  type(state_matrix_t), intent(out) :: red_state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(size(mask)) :: qn
  call state_matrix_init (red_state)
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
    qn = state_iterator_get_quantum_numbers (it)
    call quantum_numbers_undefine (qn, mask)
    call state_matrix_add_state (red_state, qn)
    call state_iterator_advance (it)
  end do
end subroutine state_matrix_reduce

```

This subroutine sets up the matrix-element array. The leaf nodes acquire the index values that point to the appropriate matrix-element entry.

We recursively scan the trie. Once we arrive at a leaf node, the index is increased and associated to that node. Finally, we allocate the matrix-element array with the appropriate size.

If matrix element values are temporarily stored within the leaf nodes, we scan the state again and transfer them to the matrix-element array.

```

<State matrices: public>+≡
public :: state_matrix_freeze

<State matrices: interfaces>≡
interface state_matrix_freeze
  module procedure state_matrix_freeze1
  module procedure state_matrix_freeze2
end interface

<State matrices: procedures>+≡
subroutine state_matrix_freeze1 (state)
  type(state_matrix_t), intent(inout), target :: state
  type(state_iterator_t) :: it
  if (associated (state%root)) then
    if (allocated (state%me)) deallocate (state%me)
    allocate (state%me (state%n_matrix_elements))
    state%me = 0
  end if
  if (state%leaf_nodes_store_values) then
    call state_iterator_init (it, state)
    do while (state_iterator_is_valid (it))
      state%me(state_iterator_get_me_index (it)) &
        = state_iterator_get_matrix_element (it)
      call state_iterator_advance (it)
    end do
    state%leaf_nodes_store_values = .false.
  end if
end subroutine state_matrix_freeze1

```

```

subroutine state_matrix_freeze2 (state)
  type(state_matrix_t), dimension(:), intent(inout), target :: state
  integer :: i
  do i = 1, size (state)
    call state_matrix_freeze1 (state(i))
  end do
end subroutine state_matrix_freeze2

```

## Direct access to the value array

Several methods for setting a value directly are summarized in this generic:

```

<State matrices: public>+≡
  public :: state_matrix_set_matrix_element

<State matrices: interfaces>+≡
  interface state_matrix_set_matrix_element
    module procedure state_matrix_set_matrix_element_qn
    module procedure state_matrix_set_matrix_element_all
    module procedure state_matrix_set_matrix_element_array
    module procedure state_matrix_set_matrix_element_single
    module procedure state_matrix_set_matrix_element_clone
  end interface

```

Set a value that corresponds to a quantum number array:

```

<State matrices: procedures>+≡
  subroutine state_matrix_set_matrix_element_qn (state, qn, value)
    type(state_matrix_t), intent(inout) :: state
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    complex(default), intent(in) :: value
    type(state_iterator_t) :: it
    if (.not. allocated (it%state%me)) then
      allocate (it%state%me (size(qn)))
    end if
    call state_iterator_init (it, state)
    do while (state_iterator_is_valid (it))
      if (all (qn == state_iterator_get_quantum_numbers (it))) then
        call state_iterator_set_matrix_element (it, value)
        return
      end if
      call state_iterator_advance (it)
    end do
  end subroutine state_matrix_set_matrix_element_qn

```

Set all matrix elements to a single value

```

<State matrices: procedures>+≡
  subroutine state_matrix_set_matrix_element_all (state, value)
    type(state_matrix_t), intent(inout) :: state
    complex(default), intent(in) :: value
    if (.not. allocated (state%me)) then
      allocate (state%me (state%n_matrix_elements))
    end if
    state%me = value
  end subroutine state_matrix_set_matrix_element_all

```

Set the matrix-element array directly.

```

<State matrices: procedures>+≡
  subroutine state_matrix_set_matrix_element_array (state, value)
    type(state_matrix_t), intent(inout) :: state
    complex(default), dimension(:), intent(in) :: value
    if (.not. allocated (state%me)) then
      allocate (state%me (size (value)))
    end if
    state%me = value
  end subroutine state_matrix_set_matrix_element_array

  pure subroutine state_matrix_set_matrix_element_single (state, i, value)
    type(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    complex(default), intent(in) :: value
    if (.not. allocated (state%me)) then
      allocate (state%me (state%n_matrix_elements))
    end if
    state%me(i) = value
  end subroutine state_matrix_set_matrix_element_single

```

Clone the matrix elements from another (matching) state matrix.

```

<State matrices: procedures>+≡
  subroutine state_matrix_set_matrix_element_clone (state, state1)
    type(state_matrix_t), intent(inout) :: state
    type(state_matrix_t), intent(in) :: state1
    if (.not. allocated (state1%me)) return
    if (.not. allocated (state%me)) allocate (state%me (size (state1%me)))
    state%me = state1%me
  end subroutine state_matrix_set_matrix_element_clone

```

Add a value to a matrix element

```

<State matrices: public>+≡
  public :: state_matrix_add_to_matrix_element

<State matrices: procedures>+≡
  subroutine state_matrix_add_to_matrix_element (state, i, value)
    type(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    complex(default), intent(in) :: value
    state%me(i) = state%me(i) + value
  end subroutine state_matrix_add_to_matrix_element

```

### 7.5.3 State iterators

Accessing the quantum state from outside is best done using a specialized iterator, i.e., a pointer to a particular branch of the quantum state trie. Technically, the iterator contains a pointer to a leaf node, but via parent pointers it allows

to access the whole branch where the leaf is attached. For quick access, we also keep the branch depth (which is assumed to be universal for a quantum state).

```

<State matrices: public>+≡
    public :: state_iterator_t

<State matrices: types>+≡
    type :: state_iterator_t
    private
        integer :: depth = 0
        type(state_matrix_t), pointer :: state => null ()
        type(node_t), pointer :: node => null ()
    end type state_iterator_t

```

The initializer: Point at the first branch. Note that this cannot be pure, thus not be elemental, because the iterator can be used to manipulate data in the state matrix.

```

<State matrices: public>+≡
    public :: state_iterator_init

<State matrices: procedures>+≡
    subroutine state_iterator_init (it, state)
        type(state_iterator_t), intent(out) :: it
        type(state_matrix_t), intent(in), target :: state
        it%state => state
        it%depth = state%depth
        if (state_matrix_is_defined (state)) then
            it%node => state%root
            do while (associated (it%node%child_first))
                it%node => it%node%child_first
            end do
        else
            it%node => null ()
        end if
    end subroutine state_iterator_init

```

Go forward. Recursively programmed: if the next node does not exist, go back to the parent node and look at its successor (if present), etc.

There is a possible pitfall in the implementation: If the dummy pointer argument to the `find_next` routine is used directly, we still get the correct result for the iterator, but calling the recursion on `node%parent` means that we manipulate a parent pointer in the original state in addition to the iterator. Making a local copy of the pointer avoids this. Using pointer intent would be helpful, but we do not yet rely on this F2003 feature.

```

<State matrices: public>+≡
    public :: state_iterator_advance

<State matrices: procedures>+≡
    subroutine state_iterator_advance (it)
        type(state_iterator_t), intent(inout) :: it
        call find_next (it%node)
    contains
        recursive subroutine find_next (node_in)
            type(node_t), intent(in), target :: node_in

```

```

type(node_t), pointer :: node
node => node_in
if (associated (node%next)) then
    node => node%next
    do while (associated (node%child_first))
        node => node%child_first
    end do
    it%node => node
else if (associated (node%parent)) then
    call find_next (node%parent)
else
    it%node => null ()
end if
end subroutine find_next
end subroutine state_iterator_advance

```

If all has been scanned, the iterator is at an undefined state. Check for this:

```

<State matrices: public>+≡
public :: state_iterator_is_valid

<State matrices: procedures>+≡
function state_iterator_is_valid (it) result (defined)
    logical :: defined
    type(state_iterator_t), intent(in) :: it
    defined = associated (it%node)
end function state_iterator_is_valid

```

Return the matrix-element index that corresponds to the current node

```

<State matrices: public>+≡
public :: state_iterator_get_me_index

<State matrices: procedures>+≡
function state_iterator_get_me_index (it) result (n)
    integer :: n
    type(state_iterator_t), intent(in) :: it
    n = it%node%me_index
end function state_iterator_get_me_index

```

Return the number of times this quantum-number state has been added (noting that it is physically inserted only the first time). Note that for each state, there is an array of counters.

```

<State matrices: public>+≡
public :: state_iterator_get_me_count

<State matrices: procedures>+≡
function state_iterator_get_me_count (it) result (n)
    integer, dimension(:), allocatable :: n
    type(state_iterator_t), intent(in) :: it
    if (allocated (it%node%me_count)) then
        allocate (n (size (it%node%me_count)))
        n = it%node%me_count
    else
        allocate (n (0))
    end if
end function state_iterator_get_me_count

```

```

        end if
    end function state_iterator_get_me_count

```

Use the iterator to retrieve quantum-number information:

```

<State matrices: public>+≡
    public :: state_iterator_get_quantum_numbers
    public :: state_iterator_get_flavor
    public :: state_iterator_get_color
    public :: state_iterator_get_helicity

<State matrices: interfaces>+≡
    interface state_iterator_get_quantum_numbers
        module procedure state_iterator_get_qn_multi
        module procedure state_iterator_get_qn_slice
        module procedure state_iterator_get_qn_range
        module procedure state_iterator_get_qn_single
    end interface

    interface state_iterator_get_flavor
        module procedure state_iterator_get_flv_multi
        module procedure state_iterator_get_flv_slice
        module procedure state_iterator_get_flv_range
        module procedure state_iterator_get_flv_single
    end interface

    interface state_iterator_get_color
        module procedure state_iterator_get_col_multi
        module procedure state_iterator_get_col_slice
        module procedure state_iterator_get_col_range
        module procedure state_iterator_get_col_single
    end interface

    interface state_iterator_get_helicity
        module procedure state_iterator_get_hel_multi
        module procedure state_iterator_get_hel_slice
        module procedure state_iterator_get_hel_range
        module procedure state_iterator_get_hel_single
    end interface

```

These versions return the whole quantum number array

```

<State matrices: procedures>+≡
    function state_iterator_get_qn_multi (it) result (qn)
        type(state_iterator_t), intent(in) :: it
        type(quantum_numbers_t), dimension(it%depth) :: qn
        type(node_t), pointer :: node
        integer :: i
        node => it%node
        do i = it%depth, 1, -1
            qn(i) = node%qn
            node => node%parent
        end do
    end function state_iterator_get_qn_multi

```



```

function state_iterator_get_flv_multi (it) result (flv)
  type(state_iterator_t), intent(in) :: it
  type(flavor_t), dimension(it%depth) :: flv
  flv = quantum_numbers_get_flavor &
    (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_flv_multi

function state_iterator_get_col_multi (it) result (col)
  type(state_iterator_t), intent(in) :: it
  type(color_t), dimension(it%depth) :: col
  col = quantum_numbers_get_color &
    (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_col_multi

function state_iterator_get_hel_multi (it) result (hel)
  type(state_iterator_t), intent(in) :: it
  type(helicity_t), dimension(it%depth) :: hel
  hel = quantum_numbers_get_helicity &
    (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_hel_multi

```

An array slice (derived from the above).

*(State matrices: procedures)*+≡

```

function state_iterator_get_qn_slice (it, index) result (qn)
  type(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(quantum_numbers_t), dimension(size(index)) :: qn
  type(quantum_numbers_t), dimension(it%depth) :: qn_tmp
  qn_tmp = state_iterator_get_qn_multi (it)
  qn = qn_tmp(index)
end function state_iterator_get_qn_slice

function state_iterator_get_flv_slice (it, index) result (flv)
  type(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(flavor_t), dimension(size(index)) :: flv
  flv = quantum_numbers_get_flavor &
    (state_iterator_get_quantum_numbers (it, index))
end function state_iterator_get_flv_slice

function state_iterator_get_col_slice (it, index) result (col)
  type(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(color_t), dimension(size(index)) :: col
  col = quantum_numbers_get_color &
    (state_iterator_get_quantum_numbers (it, index))
end function state_iterator_get_col_slice

function state_iterator_get_hel_slice (it, index) result (hel)
  type(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(helicity_t), dimension(size(index)) :: hel
  hel = quantum_numbers_get_helicity &
    (state_iterator_get_quantum_numbers (it, index))

```

```
end function state_iterator_get_hel_slice
```

An array range (implemented directly).

*(State matrices: procedures)*+≡

```
function state_iterator_get_qn_range (it, k1, k2) result (qn)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k1, k2
  type(quantum_numbers_t), dimension(k2-k1+1) :: qn
  type(node_t), pointer :: node
  integer :: i
  node => it%node
  SCAN: do i = it%depth, 1, -1
    if (k1 <= i .and. i <= k2) then
      qn(i-k1+1) = node%qn
    else
      node => node%parent
    end if
  end do SCAN
end function state_iterator_get_qn_range

function state_iterator_get_flv_range (it, k1, k2) result (flv)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k1, k2
  type(flavor_t), dimension(k2-k1+1) :: flv
  flv = quantum_numbers_get_flavor &
    (state_iterator_get_quantum_numbers (it, k1, k2))
end function state_iterator_get_flv_range

function state_iterator_get_col_range (it, k1, k2) result (col)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k1, k2
  type(color_t), dimension(k2-k1+1) :: col
  col = quantum_numbers_get_color &
    (state_iterator_get_quantum_numbers (it, k1, k2))
end function state_iterator_get_col_range

function state_iterator_get_hel_range (it, k1, k2) result (hel)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k1, k2
  type(helicity_t), dimension(k2-k1+1) :: hel
  hel = quantum_numbers_get_helicity &
    (state_iterator_get_quantum_numbers (it, k1, k2))
end function state_iterator_get_hel_range
```

Just a specific single element

*(State matrices: procedures)*+≡

```
function state_iterator_get_qn_single (it, k) result (qn)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(quantum_numbers_t) :: qn
  type(node_t), pointer :: node
  integer :: i
  node => it%node
```

```

SCAN: do i = it%depth, 1, -1
  if (i == k) then
    qn = node%qn
    exit SCAN
  else
    node => node%parent
  end if
end do SCAN
end function state_iterator_get_qn_single

function state_iterator_get_flv_single (it, k) result (flv)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(flavor_t) :: flv
  flv = quantum_numbers_get_flavor &
    (state_iterator_get_quantum_numbers (it, k))
end function state_iterator_get_flv_single

function state_iterator_get_col_single (it, k) result (col)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(color_t) :: col
  col = quantum_numbers_get_color &
    (state_iterator_get_quantum_numbers (it, k))
end function state_iterator_get_col_single

function state_iterator_get_hel_single (it, k) result (hel)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(helicity_t) :: hel
  hel = quantum_numbers_get_helicity &
    (state_iterator_get_quantum_numbers (it, k))
end function state_iterator_get_hel_single

```

Retrieve the matrix element value associated with the current node.

```

<State matrices: public>+≡
  public :: state_iterator_get_matrix_element

<State matrices: procedures>+≡
  function state_iterator_get_matrix_element (it) result (me)
    complex(default) :: me
    type(state_iterator_t), intent(in) :: it
    if (it%state%leaf_nodes_store_values) then
      me = it%node%me
    else if (it%node%me_index /= 0) then
      me = it%state%me(it%node%me_index)
    else
      me = 0
    end if
  end function state_iterator_get_matrix_element

```

Set the matrix element value using the state iterator.

```

<State matrices: public>+≡
  public :: state_iterator_set_matrix_element

```

```

<State matrices: procedures>+≡
subroutine state_iterator_set_matrix_element (it, value)
  type(state_iterator_t), intent(inout) :: it
  complex(default), intent(in) :: value
  if (it%node%me_index /= 0) then
    it%state%me(it%node%me_index) = value
  end if
end subroutine state_iterator_set_matrix_element

```

## 7.5.4 Operations on quantum states

Return a deep copy of a state matrix.

```

<State matrices: public>+≡
  public :: assignment(=)

<State matrices: interfaces>+≡
  interface assignment(=)
    module procedure state_matrix_assign
  end interface

<State matrices: procedures>+≡
subroutine state_matrix_assign (state_out, state_in)
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), intent(in), target :: state_in
  type(state_iterator_t) :: it
  if (.not. state_matrix_is_defined (state_in)) return
  call state_matrix_init (state_out)
  call state_iterator_init (it, state_in)
  do while (state_iterator_is_valid (it))
    call state_matrix_add_state (state_out, &
      state_iterator_get_quantum_numbers (it), &
      state_iterator_get_me_index (it))
    call state_iterator_advance (it)
  end do
  if (allocated (state_in%me)) then
    allocate (state_out%me (size (state_in%me)))
    state_out%me = state_in%me
  end if
end subroutine state_matrix_assign

```

Determine the indices of all diagonal matrix elements.

```

<State matrices: public>+≡
  public :: state_matrix_get_diagonal_entries

<State matrices: procedures>+≡
subroutine state_matrix_get_diagonal_entries (state, i)
  type(state_matrix_t), intent(in) :: state
  integer, dimension(:), allocatable, intent(out) :: i
  integer, dimension(state%n_matrix_elements) :: tmp
  integer :: n
  type(state_iterator_t) :: it
  n = 0

```

```

call state_iterator_init (it, state)
do while (state_iterator_is_valid (it))
  if (all (quantum_numbers_are_diagonal ( &
    state_iterator_get_quantum_numbers (it)))) then
    n = n + 1
    tmp(n) = state_iterator_get_me_index (it)
  end if
  call state_iterator_advance (it)
end do
allocate (i(n))
if (n > 0) i = tmp(:n)
end subroutine state_matrix_get_diagonal_entries

```

Normalize all matrix elements, i.e., multiply by a common factor. Assuming that the factor is nonzero, of course.

```

<State matrices: public>+≡
  public :: state_matrix_renormalize

<State matrices: procedures>+≡
  subroutine state_matrix_renormalize (state, factor)
    type(state_matrix_t), intent(inout) :: state
    complex(default), intent(in) :: factor
    state%me = state%me * factor
  end subroutine state_matrix_renormalize

```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

```

<State matrices: public>+≡
  public :: state_matrix_normalize_by_trace

<State matrices: procedures>+≡
  subroutine state_matrix_normalize_by_trace (state)
    type(state_matrix_t), intent(inout) :: state
    real(default) :: trace
    trace = state_matrix_trace (state)
    if (trace /= 0) then
      state%me = state%me / trace
      state%norm = state%norm * trace
    end if
  end subroutine state_matrix_normalize_by_trace

```

Analogous, but renormalize by maximal (absolute) value.

```

<State matrices: public>+≡
  public :: state_matrix_normalize_by_max

<State matrices: procedures>+≡
  subroutine state_matrix_normalize_by_max (state)
    type(state_matrix_t), intent(inout) :: state
    real(default) :: m
    m = maxval (abs (state%me))
    if (m /= 0) then
      state%me = state%me / m
      state%norm = state%norm * m
    end if
  end subroutine state_matrix_normalize_by_max

```

```

        end if
    end subroutine state_matrix_normalize_by_max

```

Return the sum of all matrix element values.

```

<State matrices: public>+≡
    public :: state_matrix_sum

<State matrices: procedures>+≡
    function state_matrix_sum (state) result (value)
        complex(default) :: value
        type(state_matrix_t), intent(in) :: state
        value = sum (state%me)
    end function state_matrix_sum

```

Return the trace of a state matrix, i.e., the sum over all diagonal values. If **qn\_in** is provided, only branches that match this quantum-numbers array are considered.

```

<State matrices: public>+≡
    public :: state_matrix_trace

<State matrices: procedures>+≡
    function state_matrix_trace (state, qn_in) result (trace)
        complex(default) :: trace
        type(state_matrix_t), intent(in), target :: state
        type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
        type(quantum_numbers_t), dimension(:), allocatable :: qn
        type(state_iterator_t) :: it
        allocate (qn (state_matrix_get_depth (state)))
        trace = 0
        call state_iterator_init (it, state)
        do while (state_iterator_is_valid (it))
            qn = state_iterator_get_quantum_numbers (it)
            if (present (qn_in)) then
                if (.not. all (qn .match. qn_in)) then
                    call state_iterator_advance (it); cycle
                end if
            end if
            if (all (quantum_numbers_are_diagonal (qn))) then
                trace = trace + state_iterator_get_matrix_element (it)
            end if
            call state_iterator_advance (it)
        end do
    end function state_matrix_trace

```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no **freeze** must be performed anymore.

```

<State matrices: public>+≡
    public :: state_matrix_add_color_contractions

```

```

<State matrices: procedures>+≡
subroutine state_matrix_add_color_contractions (state)
  type(state_matrix_t), intent(inout), target :: state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn_con
  integer, dimension(:), allocatable :: me_index
  integer :: depth, n_me, i, j
  depth = state_matrix_get_depth (state)
  n_me = state_matrix_get_n_matrix_elements (state)
  allocate (qn (depth, n_me))
  allocate (me_index (n_me))
  i = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
    i = i + 1
    qn(:,i) = state_iterator_get_quantum_numbers (it)
    me_index(i) = state_iterator_get_me_index (it)
    call state_iterator_advance (it)
  end do
  do i = 1, n_me
    call quantum_number_array_make_color_contractions (qn(:,i), qn_con)
    do j = 1, size (qn_con, 2)
      call state_matrix_add_state (state, qn_con(:,j), index = me_index(i))
    end do
  end do
end subroutine state_matrix_add_color_contractions

```

This procedure merges two state matrices of equal depth. For each quantum number (flavor, color, helicity), we take the entry from the first argument where defined, otherwise the second one. (If both are defined, we get an off-diagonal matrix.) The resulting trie combines the information of the input tries in all possible ways. Note that values are ignored, all values in the result are zero.

```

<State matrices: public>+≡
public :: merge_state_matrices

<State matrices: procedures>+≡
subroutine merge_state_matrices (state1, state2, state3)
  type(state_matrix_t), intent(in), target :: state1, state2
  type(state_matrix_t), intent(out) :: state3
  type(state_iterator_t) :: it1, it2
  type(quantum_numbers_t), dimension(state1%depth) :: qn1, qn2
  if (state1%depth /= state2%depth) then
    call state_matrix_write (state1)
    call state_matrix_write (state2)
    call msg_bug ("State matrices merge impossible: incompatible depths")
  end if
  call state_matrix_init (state3)
  call state_iterator_init (it1, state1)
  do while (state_iterator_is_valid (it1))
    qn1 = state_iterator_get_quantum_numbers (it1)
    call state_iterator_init (it2, state2)
    do while (state_iterator_is_valid (it2))
      qn2 = state_iterator_get_quantum_numbers (it2)

```

```

        call state_matrix_add_state &
            (state3, qn1 .merge. qn2)
        call state_iterator_advance (it2)
    end do
    call state_iterator_advance (it1)
end do
call state_matrix_freeze (state3)
end subroutine merge_state_matrices

```

Multiply matrix elements from two state matrices. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

Note that the `dot_product` intrinsic function conjugates its first complex argument. This is intended for the c suffix case, but must be reverted for the plain-product case.

*(State matrices: public)+≡*

```

public :: state_matrix_evaluate_product
public :: state_matrix_evaluate_product_cf
public :: state_matrix_evaluate_square_c
public :: state_matrix_evaluate_sum

```

*(State matrices: procedures)+≡*

```

pure subroutine state_matrix_evaluate_product &
    (state, i, state1, state2, index1, index2)
    type(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    type(state_matrix_t), intent(in) :: state1, state2
    integer, dimension(:), intent(in) :: index1, index2
    state%me(i) = &
        dot_product (conjg (state1%me(index1)), state2%me(index2))
    state%norm = state1%norm * state2%norm
end subroutine state_matrix_evaluate_product

```

```

pure subroutine state_matrix_evaluate_product_cf &
    (state, i, state1, state2, index1, index2, factor)
    type(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    type(state_matrix_t), intent(in) :: state1, state2
    integer, dimension(:), intent(in) :: index1, index2
    complex(default), dimension(:), intent(in) :: factor
    state%me(i) = &
        dot_product (state1%me(index1), factor * state2%me(index2))
    state%norm = state1%norm * state2%norm
end subroutine state_matrix_evaluate_product_cf

```

```

pure subroutine state_matrix_evaluate_square_c (state, i, state1, index1)
    type(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    type(state_matrix_t), intent(in) :: state1
    integer, dimension(:), intent(in) :: index1
    state%me(i) = &
        dot_product (state1%me(index1), state1%me(index1))

```



```

        state%norm = abs (state1%norm) ** 2
    end subroutine state_matrix_evaluate_square_c

    pure subroutine state_matrix_evaluate_sum (state, i, state1, index1)
        type(state_matrix_t), intent(inout) :: state
        integer, intent(in) :: i
        type(state_matrix_t), intent(in) :: state1
        integer, dimension(:), intent(in) :: index1
        state%me(i) = &
            sum (state1%me(index1)) * state1%norm
    end subroutine state_matrix_evaluate_sum

```

Outer product (of states and matrix elements):

```

<State matrices: public>+≡
    public :: outer_multiply

<State matrices: interfaces>+≡
    interface outer_multiply
        module procedure outer_multiply_pair
        module procedure outer_multiply_array
    end interface

```

This procedure constructs the outer product of two state matrices.

```

<State matrices: procedures>+≡
    subroutine outer_multiply_pair (state1, state2, state3)
        type(state_matrix_t), intent(in), target :: state1, state2
        type(state_matrix_t), intent(out) :: state3
        type(state_iterator_t) :: it1, it2
        type(quantum_numbers_t), dimension(state1%depth) :: qn1
        type(quantum_numbers_t), dimension(state2%depth) :: qn2
        type(quantum_numbers_t), dimension(state1%depth+state2%depth) :: qn3
        complex(default) :: val1, val2
        call state_matrix_init (state3, store_values=.true.)
        call state_iterator_init (it1, state1)
        do while (state_iterator_is_valid (it1))
            qn1 = state_iterator_get_quantum_numbers (it1)
            val1 = state_iterator_get_matrix_element (it1)
            call state_iterator_init (it2, state2)
            do while (state_iterator_is_valid (it2))
                qn2 = state_iterator_get_quantum_numbers (it2)
                val2 = state_iterator_get_matrix_element (it2)
                qn3(:state1%depth) = qn1
                qn3(state1%depth+1:) = qn2
                call state_matrix_add_state (state3, qn3, value=val1 * val2)
                call state_iterator_advance (it2)
            end do
            call state_iterator_advance (it1)
        end do
        call state_matrix_freeze (state3)
    end subroutine outer_multiply_pair

```

This executes the above routine iteratively for an arbitrary number of state matrices.

```

<State matrices: procedures>+≡
subroutine outer_multiply_array (state_in, state_out)
  type(state_matrix_t), dimension(:), intent(in), target :: state_in
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), dimension(:), allocatable, target :: state_tmp
  integer :: i, n
  n = size (state_in)
  select case (n)
  case (0)
    call state_matrix_init (state_out)
  case (1)
    state_out = state_in(1)
  case (2)
    call outer_multiply_pair (state_in(1), state_in(2), state_out)
  case default
    allocate (state_tmp (n-2))
    call outer_multiply_pair (state_in(1), state_in(2), state_tmp(1))
    do i = 2, n - 2
      call outer_multiply_pair (state_tmp(i-1), state_in(i+1), state_tmp(i))
    end do
    call outer_multiply_pair (state_tmp(n-2), state_in(n), state_out)
    call state_matrix_final (state_tmp)
  end select
end subroutine outer_multiply_array

```

### 7.5.5 Factorization

In physical events, the state matrix is factorized into single-particle state matrices. This is essentially a measurement.

In a simulation, we select one particular branch of the state matrix with a probability that is determined by the matrix elements at the leaves. (This makes sense only if the state matrix represents a squared amplitude.) The selection is based on a (random) value  $x$  between 0 and one that is provided as the third argument.

For flavor and color, we select a unique value for each particle. For polarization, we have three options (modes). Option 1 is to drop helicity information altogether and sum over all diagonal helicities. Option 2 is to select a unique diagonal helicity in the same way as flavor and color. Option 3 is, for each particle, to trace over all remaining helicities in order to obtain an array of independent single-particle helicity matrices.

Only branches that match the given quantum-number array `qn_in`, if present, are considered. For this array, color is ignored.

If the optional `correlated_state` is provided, it is assigned the correlated density matrix for the selected flavor-color branch, so multi-particle spin correlations remain available even if they are dropped in the single-particle density matrices.

The algorithm is as follows: First, we determine the normalization by summing over all diagonal matrix elements. In a second scan, we select one of the diagonal matrix elements by a cumulative comparison with the normalized random number. In the corresponding quantum number array, we undefine the

helicity entries. Then, we scan the third time. For each branch that matches the selected quantum number array (i.e., definite flavor and color, arbitrary helicity), we determine its contribution to any of the single-particle state matrices. The matrix-element value is added if all other quantum numbers are diagonal, while the helicity of the chosen particle may be arbitrary; this helicity determines the branch in the single-particle state.

As a result, flavor and color quantum numbers are selected with the correct probability. Within this subset of states, each single-particle state matrix results from tracing over all other particles. Note that the single-particle state matrices are not normalized.

The flag `ok` is set to false if the matrix element sum is zero, so factorization is not possible. This can happen if an event did not pass cuts.

```

<State matrices: parameters>≡
  integer, parameter, public :: FM_IGNORE_HELICITY = 1
  integer, parameter, public :: FM_SELECT_HELICITY = 2
  integer, parameter, public :: FM_FACTOR_HELICITY = 3

<State matrices: public>+≡
  public :: state_matrix_factorize

<State matrices: procedures>+≡
  subroutine state_matrix_factorize &
    (state, mode, x, ok, single_state, correlated_state, qn_in)
    type(state_matrix_t), intent(in), target :: state
    integer, intent(in) :: mode
    real(default), intent(in) :: x
    logical, intent(out) :: ok
    type(state_matrix_t), &
      dimension(:), allocatable, intent(out) :: single_state
    type(state_matrix_t), intent(out), optional :: correlated_state
    type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
    type(state_iterator_t) :: it
    real(default) :: s, xt
    complex(default) :: value
    integer :: i, depth
    type(quantum_numbers_t), dimension(:), allocatable :: qn, qn1
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    logical, dimension(:), allocatable :: diagonal
    logical, dimension(:, :), allocatable :: mask
    ok = .true.
    if (x /= 0) then
      xt = x * state_matrix_trace (state, qn_in)
    else
      xt = 0
    end if
    s = 0
    depth = state_matrix_get_depth (state)
    allocate (qn (depth), qn1 (depth), diagonal (depth))
    call state_iterator_init (it, state)
    do while (state_iterator_is_valid (it))
      qn = state_iterator_get_quantum_numbers (it)
      if (present (qn_in)) then
        if (.not. all (qn .fmatch. qn_in)) then

```

```

        call state_iterator_advance (it); cycle
    end if
end if
if (all (quantum_numbers_are_diagonal (qn))) then
    value = state_iterator_get_matrix_element (it)
    if (real (value, default) < 0) then
        call state_matrix_write (state)
        print *, value
        call msg_bug ("Event generation: " &
            // "Negative real part of squared matrix element value")
        value = 0
    end if
    s = s + value
    if (s > xt) exit
end if
    call state_iterator_advance (it)
end do
if (.not. state_iterator_is_valid (it)) then
    if (s == 0) ok = .false.
    call state_iterator_init (it, state)
end if
allocate (single_state (depth))
call state_matrix_init (single_state, store_values=.true.)
if (present (correlated_state)) &
    call state_matrix_init (correlated_state, store_values=.true.)
qn = state_iterator_get_quantum_numbers (it)
select case (mode)
case (FM_SELECT_HELICITY) ! single branch selected; shortcut
    do i = 1, depth
        call state_matrix_add_state (single_state(i), &
            (/qn(i)/), value=value)
    end do
    if (.not. present (correlated_state)) then
        call state_matrix_freeze (single_state)
        return
    end if
end select
allocate (qn_mask (depth))
call quantum_numbers_mask_init (qn_mask, .false., .false., .false., .true.)
call quantum_numbers_undefine (qn, qn_mask)
select case (mode)
case (FM_FACTOR_HELICITY)
    allocate (mask (depth, depth))
    mask = .false.
    forall (i = 1:depth) mask(i,i) = .true.
end select
call state_iterator_init (it, state)
do while (state_iterator_is_valid (it))
    qn1 = state_iterator_get_quantum_numbers (it)
    if (all (qn .match. qn1)) then
        diagonal = quantum_numbers_are_diagonal (qn1)
        value = state_iterator_get_matrix_element (it)
        select case (mode)
        case (FM_IGNORE_HELICITY) ! trace over diagonal states that match qn

```

```

        if (all (diagonal)) then
            do i = 1, depth
                call state_matrix_add_state (single_state(i), &
                    (/qn(i)/), value=value, sum_values=.true.)
            end do
        end if
        case (FM_FACTOR_HELICITY) ! trace over all other particles
            do i = 1, depth
                if (all (diagonal .or. mask(:,i))) then
                    call state_matrix_add_state (single_state(i), &
                        (/qn1(i)/), value=value, sum_values=.true.)
                end if
            end do
        end select
        if (present (correlated_state)) &
            call state_matrix_add_state (correlated_state, qn1, value=value)
        end if
        call state_iterator_advance (it)
    end do
    call state_matrix_freeze (single_state)
    if (present (correlated_state)) &
        call state_matrix_freeze (correlated_state)
end subroutine state_matrix_factorize

```

### 7.5.6 Test

```

<State matrices: public>+≡
public :: state_matrix_test

<State matrices: procedures>+≡
subroutine state_matrix_test ()
    print *, "State matrix test 1"
    call state_matrix_test1 ()
    print *
    print *, "State matrix test 2"
    call state_matrix_test2 ()
    print *
    print *, "State matrix test 3"
    call state_matrix_test3 ()
end subroutine state_matrix_test

```

Create two quantum states of equal depth and merge them.

```

<State matrices: procedures>+≡
subroutine state_matrix_test1 ()
    type(state_matrix_t) :: state1, state2, state3
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    call state_matrix_init (state1)
    call flavor_init (flv, (/ 1, 2, 11 /))
    call helicity_init (hel, (/ 1, 1, 1 /))

```

```

call quantum_numbers_init (qn, flv, hel)
call state_matrix_add_state (state1, qn)
call helicity_init (hel, (/ 1, 1, 1 /), (/ -1, 1, -1/))
call quantum_numbers_init (qn, flv, hel)
call state_matrix_add_state (state1, qn)
call state_matrix_freeze (state1)
call state_matrix_write (state1)
print *
call state_matrix_init (state2)
call color_init (col(1), (/ 501 /))
call color_init (col(2), (/ -501 /))
call color_init (col(3), (/ 0 /))
call helicity_init (hel, (/ -1, -1, 0 /))
call quantum_numbers_init (qn, col, hel)
call state_matrix_add_state (state2, qn)
call color_init (col(3), (/ 99 /))
call helicity_init (hel, (/ -1, -1, 0 /))
call quantum_numbers_init (qn, col, hel)
call state_matrix_add_state (state2, qn)
call state_matrix_freeze (state2)
call state_matrix_write (state2)
print *
call merge_state_matrices (state1, state2, state3)
call state_matrix_write (state3)
print *
call state_matrix_collapse (state3, &
    new_quantum_numbers_mask (.false., .false., &
        (/ .true., .false., .false. /)))
call state_matrix_write (state3)
call state_matrix_final (state1)
call state_matrix_final (state2)
call state_matrix_final (state3)
end subroutine state_matrix_test1

```

Create a correlated three-particle state matrix and factorize it.

*(State matrices: procedures)*+≡

```

subroutine state_matrix_test2
  type(state_matrix_t) :: state
  type(state_matrix_t), dimension(:), allocatable :: single_state
  type(state_matrix_t) :: correlated_state
  complex(default) :: u, val
  complex(default), dimension(-1:1) :: v
  integer :: f, h11, h12, h21, h22, i, mode
  type(flavor_t), dimension(2) :: flv
  type(color_t), dimension(2) :: col
  type(helicity_t), dimension(2) :: hel
  type(quantum_numbers_t), dimension(2) :: qn
  logical :: ok
  u = 1 / 2._default
  v(-1) = (0.6_default, 0._default)
  v( 1) = (0._default, 0.8_default)
  call state_matrix_init (state)
  do f = 1, 2
    do h11 = -1, 1, 2

```

```

do h12 = -1, 1, 2
  do h21 = -1, 1, 2
    do h22 = -1, 1, 2
      call flavor_init (flv, (/f, -f/))
      call color_init (col(1), (/ 1/))
      call color_init (col(2), (/ -1/))
      call helicity_init (hel, (/h11,h12/), (/h21, h22/))
      call quantum_numbers_init (qn, flv, col, hel)
      val = u * v(h11) * v(h12) * conjg (v(h21) * v(h22))
      call state_matrix_add_state (state, qn)
    end do
  end do
end do
end do
call state_matrix_freeze (state)
call state_matrix_write (state)
print *, "trace = ", state_matrix_trace (state)
do mode = 1, 3
  print *
  print *, "Mode = ", mode
  call state_matrix_factorize &
    (state, mode, 0.15_default, ok, single_state, correlated_state)
  do i = 1, size (single_state)
    print *
    call state_matrix_write (single_state(i))
    print *, "trace = ", state_matrix_trace (single_state(i))
  end do
  print *
  call state_matrix_write (correlated_state)
  print *, "trace = ", state_matrix_trace (correlated_state)
  call state_matrix_final (single_state)
  call state_matrix_final (correlated_state)
end do
call state_matrix_final (state)
end subroutine state_matrix_test2

```

Create a colored state matrix and add color contractions.

*<State matrices: procedures>+≡*

```

subroutine state_matrix_test3
  type(state_matrix_t) :: state
  type(flavor_t), dimension(4) :: flv
  type(color_t), dimension(4) :: col
  type(quantum_numbers_t), dimension(4) :: qn
  call state_matrix_init (state)
  call flavor_init (flv, &
    (/ 1, -HADRON_REMNANT_TRIPLET, -1, HADRON_REMNANT_TRIPLET /))
  call color_init (col(1), (/17/))
  call color_init (col(2), (/ -17/))
  call color_init (col(3), (/ -19/))
  call color_init (col(4), (/19/))
  call quantum_numbers_init (qn, flv=flv, col=col)
  call state_matrix_add_state (state, qn)
  call flavor_init (flv, &

```

```

        (/ 1, -HADRON_REMNANT_TRIPLET, 21, HADRON_REMNANT_OCTET /))
call color_init (col(1), (/17/))
call color_init (col(2), (/ -17/))
call color_init (col(3), (/3, -5/))
call color_init (col(4), (/5, -3/))
call quantum_numbers_init (qn, flv=flv, col=col)
call state_matrix_add_state (state, qn)
call state_matrix_freeze (state)
print *, "State:"
call state_matrix_write (state)
call state_matrix_add_color_contractions (state)
print *, "State with contractions:"
call state_matrix_write (state)
call state_matrix_final (state)
end subroutine state_matrix_test3

```

## 7.6 Interactions

This module defines the `interaction_t` type. It is an extension of the `state_matrix_t` type.

The state matrix is a representation of a multi-particle density matrix. It implements all possible flavor, color, and quantum-number assignments of the entries in a generic density matrix, and it can hold a complex matrix element for each entry. (Note that this matrix can hold non-diagonal entries in color and helicity space.) The `interaction_t` object associates this with a list of momenta, such that the whole object represents a multi-particle state.

The `interaction_t` holds information about which particles are incoming, virtual (i.e., kept for the records), or outgoing. Each particle can be associated to a source within another interaction. This allows us to automatically fill those interaction momenta which have been computed or defined elsewhere. It also contains internal parent-child relations and flags for (virtual) particles which are to be treated as resonances.

A quantum-number mask array summarizes, for each particle within the interaction, the treatment of flavor, color, or helicity (expose or ignore). A list of locks states which particles are bound to have an identical quantum-number mask. This is useful when the mask is changed at one place.

```

<interactions.f90>≡
  <File header>

  module interactions

    <Use kinds>
    <Use file utils>
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use sorting
    use subevents
    use expressions
    use flavors
    use colors

```



```

    use helicities
    use quantum_numbers
    use state_matrices

    <Standard module head>

    <Interactions: public>

    <Interactions: types>

    <Interactions: interfaces>

contains

    <Interactions: procedures>

end module interactions

```

### 7.6.1 External interaction links

Each particle in an interaction can have a link to a corresponding particle in another interaction. This allows to fetch the momenta of incoming or virtual particles from the interaction where they are defined. The link object consists of a pointer to the interaction and an index.

```

<Interactions: types>≡
    type :: external_link_t
    private
        type(interaction_t), pointer :: int => null ()
        integer :: i
    end type external_link_t

```

Set an external link.

```

<Interactions: procedures>≡
    subroutine external_link_set (link, int, i)
        type(external_link_t), intent(out) :: link
        type(interaction_t), target, intent(in) :: int
        integer, intent(in) :: i
        if (i /= 0) then
            link%int => int
            link%i = i
        end if
    end subroutine external_link_set

```

Reassign an external link to a new interaction (which should be an image of the original target).

```

<Interactions: procedures>+≡
    subroutine external_link_reassign (link, int_src, int_target)
        type(external_link_t), intent(inout) :: link
        type(interaction_t), intent(in) :: int_src
        type(interaction_t), intent(in), target :: int_target
        if (associated (link%int)) then
            if (link%int%tag == int_src%tag) link%int => int_target

```

```

        end if
    end subroutine external_link_reassign

```

Return true if the link is set

```

<Interactions: procedures>+=
    function external_link_is_set (link) result (flag)
        logical :: flag
        type(external_link_t), intent(in) :: link
        flag = associated (link%int)
    end function external_link_is_set

```

Return the interaction pointer.

```

<Interactions: public>=
    public :: external_link_get_ptr

<Interactions: procedures>+=
    function external_link_get_ptr (link) result (int)
        type(interaction_t), pointer :: int
        type(external_link_t), intent(in) :: link
        int => link%int
    end function external_link_get_ptr

```

Return the index within that interaction

```

<Interactions: public>+=
    public :: external_link_get_index

<Interactions: procedures>+=
    function external_link_get_index (link) result (i)
        integer :: i
        type(external_link_t), intent(in) :: link
        i = link%i
    end function external_link_get_index

```

Return a pointer to the momentum of the corresponding particle. If there is no association, return a null pointer.

```

<Interactions: procedures>+=
    function external_link_get_momentum_ptr (link) result (p)
        type(vector4_t), pointer :: p
        type(external_link_t), intent(in) :: link
        if (associated (link%int)) then
            p => link%int%p(link%i)
        else
            p => null ()
        end if
    end function external_link_get_momentum_ptr

```

## 7.6.2 Internal relations

In addition to the external links, particles within the interaction have parent-child relations. Here, more than one link is possible, and we set up a list.

```

<Interactions: types>+=

```

```

type :: internal_link_t
  private
  integer :: i
  type(internal_link_t), pointer :: next => null ()
end type internal_link_t

type :: internal_link_list_t
  private
  integer :: length = 0
  type(internal_link_t), pointer :: first => null ()
  type(internal_link_t), pointer :: last => null ()
end type internal_link_list_t

```

Add an internal link.

```

<Interactions: procedures>+=
subroutine internal_link_list_append (link_list, i)
  type(internal_link_list_t), intent(inout) :: link_list
  integer, intent(in) :: i
  type(internal_link_t), pointer :: current
  allocate (current)
  current%i = i
  if (associated (link_list%first)) then
    link_list%last%next => current
  else
    link_list%first => current
  end if
  link_list%last => current
  link_list%length = link_list%length + 1
end subroutine internal_link_list_append

```

Finalize the list of internal links.

```

<Interactions: procedures>+=
subroutine internal_link_list_final (link_list)
  type(internal_link_list_t), intent(inout) :: link_list
  type(internal_link_t), pointer :: current
  do while (associated (link_list%first))
    current => link_list%first
    link_list%first => current%next
    deallocate (current)
  end do
  link_list%last => null ()
  link_list%length = 0
end subroutine internal_link_list_final

```

We need a deep copy of this list when we assign interaction objects.

```

<Interactions: interfaces>=
interface assignment(=)
  module procedure internal_link_list_assign
end interface

<Interactions: procedures>+=
subroutine internal_link_list_assign (link_list_out, link_list_in)

```

```

type(internal_link_list_t), intent(in) :: link_list_in
type(internal_link_list_t), intent(out) :: link_list_out
type(internal_link_t), pointer :: current, copy
current => link_list_in%first
do while (associated (current))
  allocate (copy)
  copy%i = current%i
  if (associated (link_list_out%first)) then
    link_list_out%last%next => copy
  else
    link_list_out%first => copy
  end if
  link_list_out%last => copy
  current => current%next
end do
end subroutine internal_link_list_assign

```

Return true if the link list is nonempty:

```

<Interactions: procedures>+≡
function internal_link_list_has_entries (link_list) result (flag)
  logical :: flag
  type(internal_link_list_t), intent(in) :: link_list
  flag = associated (link_list%first)
end function internal_link_list_has_entries

```

Return a pointer to the first entry:

```

<Interactions: procedures>+≡
function internal_link_list_get_first_ptr (link_list) result (link)
  type(internal_link_list_t), intent(in) :: link_list
  type(internal_link_t), pointer :: link
  link => link_list%first
end function internal_link_list_get_first_ptr

```

Advance this pointer.

```

<Interactions: procedures>+≡
subroutine internal_link_advance (link)
  type(internal_link_t), pointer :: link
  link => link%next
end subroutine internal_link_advance

```

Return the index.

```

<Interactions: procedures>+≡
function internal_link_get_index (link) result (i)
  integer :: i
  type(internal_link_t), intent(in) :: link
  i = link%i
end function internal_link_get_index

```

Return the list length

```

<Interactions: procedures>+≡
function internal_link_list_get_length (link_list) result (length)

```

```

integer :: length
type(internal_link_list_t), intent(in) :: link_list
length = link_list%length
end function internal_link_list_get_length

```

### 7.6.3 The interaction type

An interaction is an entangled system of particles. Thus, the interaction object consists of two parts: the subevent, and the quantum state which technically is a trie. The subnode levels beyond the trie root node are in correspondence to the subevent, so both should be traversed in parallel.

The subevent is implemented as an allocatable array of four-momenta. The first `n_in` particles are incoming, `n_vir` particles in-between can be kept for bookkeeping, and the last `n_out` particles are outgoing.

Distinct interactions are linked by their particles: for each particle, we have the possibility of links to corresponding particles in other interactions. Furthermore, for bookkeeping purposes we have a self-link array `relations` where the parent-child relations are kept, and a flag array `resonant` which is set for an intermediate resonance.

Each momentum is associated with masks for flavor, color, and helicity. If a mask entry is set, the associated quantum number is to be ignored for that particle. If any mask has changed, the flag `update` is set.

We can have particle pairs locked together. If this is the case, the corresponding mask entries are bound to be equal. This is useful for particles that go through the interaction.

The interaction tag serves bookkeeping purposes. In particular, it identifies links in printout.

```

<Interactions: public>+≡
public :: interaction_t

<Interactions: types>+≡
type :: interaction_t
private
integer :: tag = 0
type(state_matrix_t) :: state_matrix
integer :: n_in = 0
integer :: n_vir = 0
integer :: n_out = 0
integer :: n_tot = 0
logical, dimension(:), allocatable :: p_is_known
type(vector4_t), dimension(:), allocatable :: p
type(external_link_t), dimension(:), allocatable :: source
type(internal_link_list_t), dimension(:), allocatable :: parents
type(internal_link_list_t), dimension(:), allocatable :: children
logical, dimension(:), allocatable :: resonant
type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
integer, dimension(:), allocatable :: hel_lock
logical :: update_state_matrix = .false.
logical :: update_values = .false.
end type interaction_t

```

Initialize the particle array with a fixed size. The first `n_in` particles are incoming, the rest outgoing. Masks are optional. There is also an optional tag. The interaction still needs fixing the values, but that is to be done after all branches have been added.

Interaction tags are assigned consecutively, using a `saved` variable local to this procedure. If desired, we can provide a seed for the interaction tags. Such a seed should be positive. The default seed is one. `tag=0` indicates an empty interaction.

If `set_relations` is set and true, we establish parent-child relations for all incoming and outgoing particles. Virtual particles are skipped; this option is normally used only for interactions without virtual particles.

```

<Interactions: public>+=
  public :: interaction_init

<Interactions: procedures>+=
  subroutine interaction_init &
    (int, n_in, n_vir, n_out, &
     tag, resonant, mask, hel_lock, set_relations, store_values)
    type(interaction_t), intent(out) :: int
    integer, intent(in) :: n_in, n_vir, n_out
    integer, intent(in), optional :: tag
    logical, dimension(:), intent(in), optional :: resonant
    type(quantum_numbers_mask_t), dimension(:), intent(in), optional :: mask
    integer, dimension(:), intent(in), optional :: hel_lock
    logical, intent(in), optional :: set_relations, store_values
    logical :: set_rel
    integer :: i, j
    set_rel = .false.; if (present (set_relations)) set_rel = set_relations
    call interaction_set_tag (int, tag)
    call state_matrix_init (int%state_matrix, store_values)
    int%n_in = n_in
    int%n_vir = n_vir
    int%n_out = n_out
    int%n_tot = n_in + n_vir + n_out
    allocate (int%p_is_known (int%n_tot))
    int%p_is_known = .false.
    allocate (int%p (int%n_tot))
    allocate (int%source (int%n_tot))
    allocate (int%parents (int%n_tot))
    allocate (int%children (int%n_tot))
    allocate (int%resonant (int%n_tot))
    if (present (resonant)) then
      int%resonant = resonant
    else
      int%resonant = .false.
    end if
    allocate (int%mask (int%n_tot))
    allocate (int%hel_lock (int%n_tot))
    if (present (mask)) then
      int%mask = mask
    end if
    if (present (hel_lock)) then
      int%hel_lock = hel_lock
    else

```

```

        int%hel_lock = 0
    end if
    int%update_state_matrix = .false.
    int%update_values = .true.
    if (set_rel) then
        do i = 1, n_in
            do j = 1, n_out
                call interaction_relate (int, i, n_in + j)
            end do
        end do
    end if
end subroutine interaction_init

```

Set or create a unique tag for the interaction. Without interaction, reset the tag counter.

*(Interactions: procedures)*+≡

```

subroutine interaction_set_tag (int, tag)
    type(interaction_t), intent(inout), optional :: int
    integer, intent(in), optional :: tag
    integer, save :: stored_tag = 1
    if (present (int)) then
        if (present (tag)) then
            int%tag = tag
        else
            int%tag = stored_tag
            stored_tag = stored_tag + 1
        end if
    else if (present (tag)) then
        stored_tag = tag
    else
        stored_tag = 1
    end if
end subroutine interaction_set_tag

```

The public interface for the previous procedure only covers the reset functionality.

*(Interactions: public)*+≡

```

public :: reset_interaction_counter

```

*(Interactions: procedures)*+≡

```

subroutine reset_interaction_counter (tag)
    integer, intent(in), optional :: tag
    call interaction_set_tag (tag=tag)
end subroutine reset_interaction_counter

```

Finalizer: The state-matrix object contains pointers.

*(Interactions: public)*+≡

```

public :: interaction_final

```

*(Interactions: procedures)*+≡

```

elemental subroutine interaction_final (int)
    type(interaction_t), intent(inout) :: int
    call state_matrix_final (int%state_matrix)

```

```
end subroutine interaction_final
```

Output. The `verbose` option refers to the state matrix output.

*(Interactions: public)*+≡

```
public :: interaction_write
```

*(Interactions: procedures)*+≡

```
subroutine interaction_write &
  (int, unit, verbose, show_momentum_sum, show_mass, show_state)
  type(interaction_t), intent(in) :: int
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
  logical, intent(in), optional :: show_state
  integer :: u
  integer :: i, index_link
  type(internal_link_t), pointer :: link
  type(interaction_t), pointer :: int_link
  logical :: show_st
  u = output_unit (unit); if (u < 0) return
  show_st = .true.; if (present (show_state)) show_st = show_state
  if (int%tag /= 0) then
    write (u, "(1x,A,I0)") "Interaction: ", int%tag
    do i = 1, int%n_tot
      if (i == 1 .and. int%n_in > 0) then
        write (u, "(1x,A)") "Incoming:"
      else if (i == int%n_in + 1 .and. int%n_vir > 0) then
        write (u, "(1x,A)") "Virtual:"
      else if (i == int%n_in + int%n_vir + 1 .and. int%n_out > 0) then
        write (u, "(1x,A)") "Outgoing:"
      end if
      write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
      if (allocated (int%resonant)) then
        if (int%resonant(i)) then
          write (u, "(A)") "[r]"
        else
          write (u, *)
        end if
      else
        write (u, *)
      end if
      if (allocated (int%p)) then
        if (int%p_is_known(i)) then
          call vector4_write (int%p(i), u, show_mass)
        else
          write (u, "(A)") " [momentum undefined]"
        end if
      else
        write (u, "(A)") " [momentum not allocated]"
      end if
      if (allocated (int%mask)) then
        write (u, "(1x,A)", advance="no") "mask [fch] = "
        call quantum_numbers_mask_write (int%mask(i), u)
        write (u, *)
      end if
    end do
  end if
```



```

if (internal_link_list_has_entries (int%parents(i)) &
    .or. internal_link_list_has_entries (int%children(i))) then
write (u, "(1x,A)", advance="no") "internal links:"
link => internal_link_list_get_first_ptr (int%parents(i))
do while (associated (link))
    write (u, "(1x,I0)", advance="no") &
        internal_link_get_index (link)
    call internal_link_advance (link)
end do
if (internal_link_list_has_entries (int%parents(i))) &
    write (u, "(1x,A)", advance="no") "=>"
write (u, "(1x,A)", advance="no") "X"
if (internal_link_list_has_entries (int%children(i))) &
    write (u, "(1x,A)", advance="no") "=>"
link => internal_link_list_get_first_ptr (int%children(i))
do while (associated (link))
    write (u, "(1x,I0)", advance="no") &
        internal_link_get_index (link)
    call internal_link_advance (link)
end do
write (u, *)
end if
if (allocated (int%hel_lock)) then
if (int%hel_lock(i) /= 0) then
    write (u, "(1x,A,1x,I0)") "helicity lock:", int%hel_lock(i)
end if
end if
if (external_link_is_set (int%source(i))) then
write (u, "(1x,A)", advance="no") "source:"
int_link => external_link_get_ptr (int%source(i))
index_link = external_link_get_index (int%source(i))
write (u, "(1x,'(,I0,')',I0)", advance="no") &
    int_link%tag, index_link
write (u, *)
end if
end do
if (present (show_momentum_sum)) then
if (allocated (int%p) .and. show_momentum_sum) then
    write (u, "(1x,A)") "Incoming particles (sum):"
    call vector4_write &
        (sum (int%p(1:int%n_in)), u, show_mass)
    write (u, "(1x,A)") "Outgoing particles (sum):"
    call vector4_write &
        (sum (int%p(int%n_in+int%n_vir+1:)), u, show_mass)
    write (u, *)
end if
end if
if (show_st) then
    call state_matrix_write (int%state_matrix, &
        write_value_list=verbose, verbose=verbose, unit=unit)
end if
else
    write (u, "(1x,A)") "Interaction: [empty]"
end if

```

```
end subroutine interaction_write
```

Assignment: We implement this as a deep copy. This applies, in particular, to the state-matrix and internal-link components. Furthermore, the new interaction acquires a new tag.

```

<Interactions: public>+≡
  public :: assignment(=)

<Interactions: interfaces>+≡
  interface assignment(=)
    module procedure interaction_assign
  end interface

<Interactions: procedures>+≡
  subroutine interaction_assign (int_out, int_in)
    type(interaction_t), intent(out) :: int_out
    type(interaction_t), intent(in), target :: int_in
    call interaction_set_tag (int_out)
    int_out%state_matrix = int_in%state_matrix
    int_out%n_in = int_in%n_in
    int_out%n_out = int_in%n_out
    int_out%n_vir = int_in%n_vir
    int_out%n_tot = int_in%n_tot
    if (allocated (int_in%p_is_known)) then
      allocate (int_out%p_is_known (size (int_in%p_is_known)))
      int_out%p_is_known = int_in%p_is_known
    end if
    if (allocated (int_in%p)) then
      allocate (int_out%p (size (int_in%p)))
      int_out%p = int_in%p
    end if
    if (allocated (int_in%source)) then
      allocate (int_out%source (size (int_in%source)))
      int_out%source = int_in%source
    end if
    if (allocated (int_in%parents)) then
      allocate (int_out%parents (size (int_in%parents)))
      int_out%parents = int_in%parents
    end if
    if (allocated (int_in%children)) then
      allocate (int_out%children (size (int_in%children)))
      int_out%children = int_in%children
    end if
    if (allocated (int_in%resonant)) then
      allocate (int_out%resonant (size (int_in%resonant)))
      int_out%resonant = int_in%resonant
    end if
    if (allocated (int_in%mask)) then
      allocate (int_out%mask (size (int_in%mask)))
      int_out%mask = int_in%mask
    end if
    if (allocated (int_in%hel_lock)) then
      allocate (int_out%hel_lock (size (int_in%hel_lock)))

```

```

        int_out%hel_lock = int_in%hel_lock
    end if
    int_out%update_state_matrix = int_in%update_state_matrix
    int_out%update_values = int_in%update_values
end subroutine interaction_assign

```

#### 7.6.4 Methods inherited from the state matrix member

Until F2003 is standard, we cannot implement inheritance directly. Therefore, we need wrappers for “inherited” methods.

Make a new branch in the state matrix if it does not yet exist. This is not just a wrapper but it introduces the interaction mask: where a quantum number is masked, it is not transferred but set undefined. After this, the value array has to be updated.

```

<Interactions: public>+≡
    public :: interaction_add_state

<Interactions: procedures>+≡
    subroutine interaction_add_state &
        (int, qn, index, value, sum_values, counter_index, me_index)
        type(interaction_t), intent(inout) :: int
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        integer, intent(in), optional :: index
        complex(default), intent(in), optional :: value
        logical, intent(in), optional :: sum_values
        integer, intent(in), optional :: counter_index
        integer, intent(out), optional :: me_index
        type(quantum_numbers_t), dimension(size(qn)) :: qn_tmp
        qn_tmp = qn
        call quantum_numbers_undefine (qn_tmp, int%mask)
        call state_matrix_add_state &
            (int%state_matrix, qn_tmp, index, value, sum_values, &
             counter_index, me_index)
        int%update_values = .true.
    end subroutine interaction_add_state

```

Freeze the quantum state: First collapse the quantum state, i.e., remove quantum numbers if any mask has changed, then fix the array of value pointers.

```

<Interactions: public>+≡
    public :: interaction_freeze

<Interactions: procedures>+≡
    subroutine interaction_freeze (int)
        type(interaction_t), intent(inout) :: int
        if (int%update_state_matrix) then
            call state_matrix_collapse (int%state_matrix, int%mask)
            int%update_state_matrix = .false.
            int%update_values = .true.
        end if
        if (int%update_values) then
            call state_matrix_freeze (int%state_matrix)
            int%update_values = .false.
        end if
    end subroutine interaction_freeze

```

```

        end if
    end subroutine interaction_freeze

```

Return true if the state matrix is empty.

```

<Interactions: public>+≡
    public :: interaction_is_empty

<Interactions: procedures>+≡
    function interaction_is_empty (int) result (flag)
        logical :: flag
        type(interaction_t), intent(in) :: int
        flag = state_matrix_is_empty (int%state_matrix)
    end function interaction_is_empty

```

Get the number of values stored in the state matrix:

```

<Interactions: public>+≡
    public :: interaction_get_n_matrix_elements

<Interactions: procedures>+≡
    function interaction_get_n_matrix_elements (int) result (n)
        integer :: n
        type(interaction_t), intent(in) :: int
        n = state_matrix_get_n_matrix_elements (int%state_matrix)
    end function interaction_get_n_matrix_elements

```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

```

<Interactions: public>+≡
    public :: interaction_get_norm

<Interactions: procedures>+≡
    function interaction_get_norm (int) result (norm)
        real(default) :: norm
        type(interaction_t), intent(in) :: int
        norm = state_matrix_get_norm (int%state_matrix)
    end function interaction_get_norm

```

Get the quantum number array that corresponds to a given index.

```

<Interactions: public>+≡
    public :: interaction_get_quantum_numbers

<Interactions: procedures>+≡
    function interaction_get_quantum_numbers (int, i) result (qn)
        type(quantum_numbers_t), dimension(:), allocatable :: qn
        type(interaction_t), intent(in), target :: int
        integer, intent(in) :: i
        allocate (qn (state_matrix_get_depth (int%state_matrix)))
        qn = state_matrix_get_quantum_numbers (int%state_matrix, i)
    end function interaction_get_quantum_numbers

```

Get the matrix element that corresponds to a set of quantum numbers, a given index, or return the whole array.

```

<Interactions: public>+≡
    public :: interaction_get_matrix_element

```

```

<Interactions: procedures>+≡
function interaction_get_matrix_element (int, i) result (me)
  complex(default) :: me
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  me = state_matrix_get_matrix_element (int%state_matrix, i)
end function interaction_get_matrix_element

```

Set the complex value(s) stored in the quantum state.

```

<Interactions: public>+≡
public :: interaction_set_matrix_element

<Interactions: interfaces>+≡
interface interaction_set_matrix_element
  module procedure interaction_set_matrix_element_qn
  module procedure interaction_set_matrix_element_all
  module procedure interaction_set_matrix_element_array
  module procedure interaction_set_matrix_element_single
  module procedure interaction_set_matrix_element_clone
end interface

```

Indirect access via the quantum number array:

```

<Interactions: procedures>+≡
subroutine interaction_set_matrix_element_qn (int, qn, val)
  type(interaction_t), intent(inout) :: int
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  complex(default), intent(in) :: val
  call state_matrix_set_matrix_element (int%state_matrix, qn, val)
end subroutine interaction_set_matrix_element_qn

```

Set all entries of the matrix-element array to a given value.

```

<Interactions: procedures>+≡
subroutine interaction_set_matrix_element_all (int, value)
  type(interaction_t), intent(inout) :: int
  complex(default), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, value)
end subroutine interaction_set_matrix_element_all

```

Set the matrix-element array directly.

```

<Interactions: procedures>+≡
subroutine interaction_set_matrix_element_array (int, value)
  type(interaction_t), intent(inout) :: int
  complex(default), dimension(:), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, value)
end subroutine interaction_set_matrix_element_array

pure subroutine interaction_set_matrix_element_single (int, i, value)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  complex(default), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, i, value)
end subroutine interaction_set_matrix_element_single

```

Clone from another (matching) interaction.

```
<Interactions: procedures>+≡  
  subroutine interaction_set_matrix_element_clone (int, int1)  
    type(interaction_t), intent(inout) :: int  
    type(interaction_t), intent(in) :: int1  
    call state_matrix_set_matrix_element (int%state_matrix, int1%state_matrix)  
  end subroutine interaction_set_matrix_element_clone
```

Get the indices of any diagonal matrix elements.

```
<Interactions: public>+≡  
  public :: interaction_get_diagonal_entries  
  
<Interactions: procedures>+≡  
  subroutine interaction_get_diagonal_entries (int, i)  
    type(interaction_t), intent(in) :: int  
    integer, dimension(:), allocatable, intent(out) :: i  
    call state_matrix_get_diagonal_entries (int%state_matrix, i)  
  end subroutine interaction_get_diagonal_entries
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

```
<Interactions: public>+≡  
  public :: interaction_normalize_by_trace  
  
<Interactions: procedures>+≡  
  subroutine interaction_normalize_by_trace (int)  
    type(interaction_t), intent(inout) :: int  
    call state_matrix_normalize_by_trace (int%state_matrix)  
  end subroutine interaction_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

```
<Interactions: public>+≡  
  public :: interaction_normalize_by_max  
  
<Interactions: procedures>+≡  
  subroutine interaction_normalize_by_max (int)  
    type(interaction_t), intent(inout) :: int  
    call state_matrix_normalize_by_max (int%state_matrix)  
  end subroutine interaction_normalize_by_max
```

Return the maximum absolute value of color indices.

```
<Interactions: public>+≡  
  public :: interaction_get_max_color_value  
  
<Interactions: procedures>+≡  
  function interaction_get_max_color_value (int) result (cmax)  
    integer :: cmax  
    type(interaction_t), intent(in) :: int  
    cmax = state_matrix_get_max_color_value (int%state_matrix)  
  end function interaction_get_max_color_value
```

Factorize the state matrix into single-particle state matrices, the branch selection depending on a (random) value between 0 and 1; optionally also return a correlated state matrix.

```

<Interactions: public>+≡
    public :: interaction_factorize
<Interactions: procedures>+≡
    subroutine interaction_factorize &
        (int, mode, x, ok, single_state, correlated_state, qn_in)
        type(interaction_t), intent(in), target :: int
        integer, intent(in) :: mode
        real(default), intent(in) :: x
        logical, intent(out) :: ok
        type(state_matrix_t), &
            dimension(:), allocatable, intent(out) :: single_state
        type(state_matrix_t), intent(out), optional :: correlated_state
        type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
        call state_matrix_factorize &
            (int%state_matrix, mode, x, ok, single_state, correlated_state, qn_in)
    end subroutine interaction_factorize

```

Sum all matrix element values

```

<Interactions: public>+≡
    public :: interaction_sum
<Interactions: procedures>+≡
    function interaction_sum (int) result (value)
        complex(default) :: value
        type(interaction_t), intent(in) :: int
        value = state_matrix_sum (int%state_matrix)
    end function interaction_sum

```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no `freeze` must be performed anymore.

```

<Interactions: public>+≡
    public :: interaction_add_color_contractions
<Interactions: procedures>+≡
    subroutine interaction_add_color_contractions (int)
        type(interaction_t), intent(inout) :: int
        call state_matrix_add_color_contractions (int%state_matrix)
    end subroutine interaction_add_color_contractions

```

Multiply matrix elements from two interactions. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

```

<Interactions: public>+≡
    public :: interaction_evaluate_product
    public :: interaction_evaluate_product_cf
    public :: interaction_evaluate_square_c
    public :: interaction_evaluate_sum

```

```

<Interactions: procedures>+≡
  pure subroutine interaction_evaluate_product &
    (int, i, int1, int2, index1, index2)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(interaction_t), intent(in) :: int1, int2
    integer, dimension(:), intent(in) :: index1, index2
    call state_matrix_evaluate_product &
      (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
        index1, index2)
  end subroutine interaction_evaluate_product

  pure subroutine interaction_evaluate_product_cf &
    (int, i, int1, int2, index1, index2, factor)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(interaction_t), intent(in) :: int1, int2
    integer, dimension(:), intent(in) :: index1, index2
    complex(default), dimension(:), intent(in) :: factor
    call state_matrix_evaluate_product_cf &
      (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
        index1, index2, factor)
  end subroutine interaction_evaluate_product_cf

  pure subroutine interaction_evaluate_square_c (int, i, int1, index1)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(interaction_t), intent(in) :: int1
    integer, dimension(:), intent(in) :: index1
    call state_matrix_evaluate_square_c &
      (int%state_matrix, i, int1%state_matrix, index1)
  end subroutine interaction_evaluate_square_c

  pure subroutine interaction_evaluate_sum (int, i, int1, index1)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(interaction_t), intent(in) :: int1
    integer, dimension(:), intent(in) :: index1
    call state_matrix_evaluate_sum &
      (int%state_matrix, i, int1%state_matrix, index1)
  end subroutine interaction_evaluate_sum

```

### 7.6.5 Accessing contents

Return the integer tag.

```

<Interactions: public>+≡
  public :: interaction_get_tag

<Interactions: procedures>+≡
  function interaction_get_tag (int) result (tag)
    integer :: tag
    type(interaction_t), intent(in) :: int
    tag = int%tag

```



```
end function interaction_get_tag
```

Return the number of particles.

*(Interactions: public)+≡*

```
public :: interaction_get_n_tot
public :: interaction_get_n_in
public :: interaction_get_n_vir
public :: interaction_get_n_out
```

*(Interactions: procedures)+≡*

```
function interaction_get_n_tot (int) result (n_tot)
  integer :: n_tot
  type(interaction_t), intent(in) :: int
  n_tot = int%n_tot
end function interaction_get_n_tot
```

```
function interaction_get_n_in (int) result (n_in)
  integer :: n_in
  type(interaction_t), intent(in) :: int
  n_in = int%n_in
end function interaction_get_n_in
```

```
function interaction_get_n_vir (int) result (n_vir)
  integer :: n_vir
  type(interaction_t), intent(in) :: int
  n_vir = int%n_vir
end function interaction_get_n_vir
```

```
function interaction_get_n_out (int) result (n_out)
  integer :: n_out
  type(interaction_t), intent(in) :: int
  n_out = int%n_out
end function interaction_get_n_out
```

Return a momentum index. The flags specify whether to keep/drop incoming, virtual, or outgoing momenta. Check for illegal values.

*(Interactions: procedures)+≡*

```
function idx (int, i, outgoing)
  integer :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  logical :: in, vir, out
  if (present (outgoing)) then
    in = .not. outgoing
    vir = .false.
    out = outgoing
  else
    in = .true.
    vir = .true.
    out = .true.
  end if
  idx = 0
```

```

if (in) then
  if (vir) then
    if (out) then
      if (i <= int%n_tot) idx = i
    else
      if (i <= int%n_in + int%n_vir) idx = i
    end if
  else if (out) then
    if (i <= int%n_in) then
      idx = i
    else if (i <= int%n_in + int%n_out) then
      idx = int%n_vir + i
    end if
  else
    if (i <= int%n_in) idx = i
  end if
else if (vir) then
  if (out) then
    if (i <= int%n_vir + int%n_out) idx = int%n_in + i
  else
    if (i <= int%n_vir) idx = int%n_in + i
  end if
else if (out) then
  if (i <= int%n_out) idx = int%n_in + int%n_vir + i
end if
if (idx == 0) then
  call interaction_write (int)
  print *, i, in, vir, out
  call msg_bug (" Momentum index is out of range for this interaction")
end if
end function idx

```

Return all or just a specific four-momentum.

```

<Interactions: public>+≡
  public :: interaction_get_momenta
  public :: interaction_get_momentum

<Interactions: interfaces>+≡
  interface interaction_get_momenta
    module procedure interaction_get_momenta_all
    module procedure interaction_get_momenta_idx
  end interface

<Interactions: procedures>+≡
  function interaction_get_momenta_all (int, outgoing) result (p)
    type(vector4_t), dimension(:), allocatable :: p
    type(interaction_t), intent(in) :: int
    logical, intent(in), optional :: outgoing
    integer :: i
    if (present (outgoing)) then
      if (outgoing) then
        allocate (p (int%n_out))
      else
        allocate (p (int%n_in))
      end if
    end if
  end function

```

```

    else
        allocate (p (int%n_tot))
    end if
    do i = 1, size (p)
        p(i) = int%p(idx (int, i, outgoing))
    end do
end function interaction_get_momenta_all

function interaction_get_momenta_idx (int, jj) result (p)
    type(vector4_t), dimension(:), allocatable :: p
    type(interaction_t), intent(in) :: int
    integer, dimension(:), intent(in) :: jj
    allocate (p (size (jj)))
    p = int%p(jj)
end function interaction_get_momenta_idx

function interaction_get_momentum (int, i, outgoing) result (p)
    type(vector4_t) :: p
    type(interaction_t), intent(in) :: int
    integer, intent(in) :: i
    logical, intent(in), optional :: outgoing
    p = int%p(idx (int, i, outgoing))
end function interaction_get_momentum

```

This is a variant as a subroutine. Redundant, but the function above fails at times for gfortran 4.5.0 (double allocation, compiler bug).

```

<Interactions: public>+≡
    public :: interaction_get_momenta_sub

<Interactions: procedures>+≡
    subroutine interaction_get_momenta_sub (int, p, outgoing)
        type(vector4_t), dimension(:), intent(out) :: p
        type(interaction_t), intent(in) :: int
        logical, intent(in), optional :: outgoing
        integer :: i
        do i = 1, size (p)
            p(i) = int%p(idx (int, i, outgoing))
        end do
    end subroutine interaction_get_momenta_sub

```

Transfer PDG codes, masses (initialization) and momenta to a predefined subevent. We use the flavor assignment of the first branch in the interaction state matrix. Only incoming and outgoing particles are transferred. Switch momentum sign for incoming particles.

```

<Interactions: public>+≡
    public :: interaction_to_subevt
    public :: interaction_momenta_to_subevt

<Interactions: interfaces>+≡
    interface interaction_momenta_to_subevt
        module procedure interaction_momenta_to_subevt_id
        module procedure interaction_momenta_to_subevt_tr
    end interface

```

*(Interactions: procedures)*+≡

```

subroutine interaction_to_subevt (int, j_beam, j_in, j_out, subevt)
  type(interaction_t), intent(in), target :: int
  integer, dimension(:), intent(in) :: j_beam, j_in, j_out
  type(subevt_t), intent(out) :: subevt
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: n_beam, n_in, n_out, i, j
  allocate (flv (int%n_tot))
  flv = quantum_numbers_get_flavor (interaction_get_quantum_numbers (int, 1))
  n_beam = size (j_beam)
  n_in = size (j_in)
  n_out = size (j_out)
  call subevt_init (subevt, n_beam + n_in + n_out)
  do i = 1, n_beam
    j = j_beam(i)
    call subevt_set_beam (subevt, i, &
      flavor_get_pdg (flv(j)), &
      vector4_null, &
      flavor_get_mass (flv(j)) ** 2)
  end do
  do i = 1, n_in
    j = j_in(i)
    call subevt_set_incoming (subevt, n_beam + i, &
      flavor_get_pdg (flv(j)), &
      vector4_null, &
      flavor_get_mass (flv(j)) ** 2)
  end do
  do i = 1, n_out
    j = j_out(i)
    call subevt_set_outgoing (subevt, n_beam + n_in + i, &
      flavor_get_pdg (flv(j)), &
      vector4_null, &
      flavor_get_mass (flv(j)) ** 2)
  end do
end subroutine interaction_to_subevt

subroutine interaction_momenta_to_subevt_id (int, j_beam, j_in, j_out, subevt)
  type(interaction_t), intent(in) :: int
  integer, dimension(:), intent(in) :: j_beam, j_in, j_out
  type(subevt_t), intent(inout) :: subevt
  call subevt_set_p_beam &
    (subevt, - interaction_get_momenta (int, j_beam))
  call subevt_set_p_incoming &
    (subevt, - interaction_get_momenta (int, j_in))
  call subevt_set_p_outgoing &
    (subevt, interaction_get_momenta (int, j_out))
end subroutine interaction_momenta_to_subevt_id

subroutine interaction_momenta_to_subevt_tr &
  (int, j_beam, j_in, j_out, lt, subevt)
  type(interaction_t), intent(in) :: int
  integer, dimension(:), intent(in) :: j_beam, j_in, j_out
  type(subevt_t), intent(inout) :: subevt
  type(lorentz_transformation_t), intent(in) :: lt

```

```

call subevt_set_p_beam &
  (subevt, - lt * interaction_get_momenta (int, j_beam))
call subevt_set_p_incoming &
  (subevt, - lt * interaction_get_momenta (int, j_in))
call subevt_set_p_outgoing &
  (subevt, lt * interaction_get_momenta (int, j_out))
end subroutine interaction_momenta_to_subevt_tr

```

Return a shallow copy of the state matrix:

```

<Interactions: public>+≡
  public :: interaction_get_state_matrix_ptr

<Interactions: procedures>+≡
  function interaction_get_state_matrix_ptr (int) result (state)
    type(state_matrix_t), pointer :: state
    type(interaction_t), intent(in), target :: int
    state => int%state_matrix
  end function interaction_get_state_matrix_ptr

```

Return the array of resonance flags

```

<Interactions: public>+≡
  public :: interaction_get_resonance_flags

<Interactions: procedures>+≡
  function interaction_get_resonance_flags (int) result (resonant)
    type(interaction_t), intent(in) :: int
    logical, dimension(size(int%resonant)) :: resonant
    resonant = int%resonant
  end function interaction_get_resonance_flags

```

Return the quantum-numbers mask (or part of it)

```

<Interactions: public>+≡
  public :: interaction_get_mask

<Interactions: interfaces>+≡
  interface interaction_get_mask
    module procedure interaction_get_mask_all
    module procedure interaction_get_mask_slice
  end interface

<Interactions: procedures>+≡
  function interaction_get_mask_all (int) result (mask)
    type(interaction_t), intent(in) :: int
    type(quantum_numbers_mask_t), dimension(size(int%mask)) :: mask
    mask = int%mask
  end function interaction_get_mask_all

  function interaction_get_mask_slice (int, index) result (mask)
    type(interaction_t), intent(in) :: int
    integer, dimension(:), intent(in) :: index
    type(quantum_numbers_mask_t), dimension(size(index)) :: mask
    mask = int%mask(index)
  end function interaction_get_mask_slice

```

Compute the invariant mass squared of the incoming particles (if any, otherwise outgoing).

```

<Interactions: public>+=
    public :: interaction_get_s

<Interactions: procedures>+=
    function interaction_get_s (int) result (s)
        real(default) :: s
        type(interaction_t), intent(in) :: int
        if (int%n_in /= 0) then
            s = sum (int%p(:int%n_in)) ** 2
        else
            s = sum (int%p(int%n_vir+1:)) ** 2
        end if
    end function interaction_get_s

```

Compute the Lorentz transformation that transforms the incoming particles from the center-of-mass frame to the lab frame where they are given. If the c.m. mass squared is negative or zero, return the identity.

```

<Interactions: public>+=
    public :: interaction_get_cm_transformation

<Interactions: procedures>+=
    function interaction_get_cm_transformation (int) result (lt)
        type(lorentz_transformation_t) :: lt
        type(interaction_t), intent(in) :: int
        type(vector4_t) :: p_cm
        real(default) :: s
        if (int%n_in /= 0) then
            p_cm = sum (int%p(:int%n_in))
        else
            p_cm = sum (int%p(int%n_vir+1:))
        end if
        s = p_cm ** 2
        if (s > 0) then
            lt = boost (p_cm, sqrt (s))
        else
            lt = identity
        end if
    end function interaction_get_cm_transformation

```

Return flavor, momentum, and position of the first outgoing unstable particle present in the interaction. Note that we need not iterate through the state matrix; if there is an unstable particle, it will be present in all state-matrix entries.

```

<Interactions: public>+=
    public :: interaction_get_unstable_particle

<Interactions: procedures>+=
    subroutine interaction_get_unstable_particle (int, flv, p, i)
        type(interaction_t), intent(in), target :: int
        type(flavor_t), intent(out) :: flv
        type(vector4_t), intent(out) :: p
        integer, intent(out) :: i
    end subroutine interaction_get_unstable_particle

```

```

type(state_iterator_t) :: it
type(flavor_t), dimension(int%n_tot) :: flv_array
call state_iterator_init (it, int%state_matrix)
flv_array = state_iterator_get_flavor (it)
do i = int%n_in + int%n_vir + 1, int%n_tot
  if (.not. flavor_is_stable (flv_array(i))) then
    flv = flv_array(i)
    p = int%p(i)
    return
  end if
end do
end subroutine interaction_get_unstable_particle

```

### 7.6.6 Modifying contents

Set the quantum numbers mask.

```

<Interactions: public>+≡
  public :: interaction_set_mask

<Interactions: procedures>+≡
  subroutine interaction_set_mask (int, mask)
    type(interaction_t), intent(inout) :: int
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    int%mask = mask
    int%update_state_matrix = .true.
  end subroutine interaction_set_mask

```

Merge a particular mask entry, respecting a possible helicity lock for this entry. We apply an OR relation, which means that quantum numbers are summed over if either of the two masks requires it.

```

<Interactions: procedures>+≡
  subroutine interaction_merge_mask_entry (int, i, mask)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(quantum_numbers_mask_t), intent(in) :: mask
    type(quantum_numbers_mask_t) :: mask_tmp
    integer :: ii
    ii = idx (int, i)
    if (int%mask(ii) .neqv. mask) then
      int%mask(ii) = int%mask(ii) .or. mask
      if (int%hel_lock(ii) /= 0) then
        call quantum_numbers_mask_assign (mask_tmp, mask, helicity=.true.)
        int%mask(int%hel_lock(ii)) = int%mask(int%hel_lock(ii)) .or. mask_tmp
      end if
    end if
    int%update_state_matrix = .true.
  end subroutine interaction_merge_mask_entry

```

Fill the momenta array, do not care about the quantum numbers of particles.

```

<Interactions: public>+≡
  public :: interaction_reset_momenta

```

```

public :: interaction_set_momenta
public :: interaction_set_momentum

<Interactions: procedures>+=
subroutine interaction_reset_momenta (int)
  type(interaction_t), intent(inout) :: int
  int%p = vector4_null
  int%p_is_known = .true.
end subroutine interaction_reset_momenta

subroutine interaction_set_momenta (int, p, outgoing)
  type(interaction_t), intent(inout) :: int
  type(vector4_t), dimension(:), intent(in) :: p
  logical, intent(in), optional :: outgoing
  integer :: i, index
  do i = 1, size (p)
    index = idx (int, i, outgoing)
    int%p(index) = p(i)
    int%p_is_known(index) = .true.
  end do
end subroutine interaction_set_momenta

subroutine interaction_set_momentum (int, p, i, outgoing)
  type(interaction_t), intent(inout) :: int
  type(vector4_t), intent(in) :: p
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  integer :: index
  index = idx (int, i, outgoing)
  int%p(index) = p
  int%p_is_known(index) = .true.
end subroutine interaction_set_momentum

```

This more sophisticated version of setting values is used for structure functions, in particular if nontrivial flavor, color, and helicity may be present: set values selectively for the given flavors. If there is more than one flavor, scan the interaction and check for a matching flavor at the specified particle location. If it matches, insert the value that corresponds to this flavor.

```

<Interactions: public>+=
public :: interaction_set_flavored_values

<Interactions: procedures>+=
subroutine interaction_set_flavored_values (int, value, flv_in, pos)
  type(interaction_t), intent(inout) :: int
  complex(default), dimension(:), intent(in) :: value
  type(flavor_t), dimension(:), intent(in) :: flv_in
  integer, intent(in) :: pos
  type(state_iterator_t) :: it
  type(flavor_t) :: flv
  integer :: i
  if (size (value) == 1) then
    call interaction_set_matrix_element (int, value(1))
  else
    call state_iterator_init (it, int%state_matrix)
  end if
end subroutine interaction_set_flavored_values

```



```

do while (state_iterator_is_valid (it))
  flv = state_iterator_get_flavor (it, pos)
  SCAN_FLV: do i = 1, size (value)
    if (flv == flv_in(i)) then
      call state_iterator_set_matrix_element (it, value(i))
      exit SCAN_FLV
    end if
  end do SCAN_FLV
  call state_iterator_advance (it)
end do
end if
end subroutine interaction_set_flavored_values

```

### 7.6.7 Handling Linked interactions

Store relations between corresponding particles within one interaction. The first particle is the parent, the second one the child. Links are established in both directions.

These relations have no effect on the propagation of momenta etc., they are rather used for mother-daughter relations in event output.

*<Interactions: public>+≡*

```
public :: interaction_relate
```

*<Interactions: procedures>+≡*

```

subroutine interaction_relate (int, i1, i2)
  type(interaction_t), intent(inout), target :: int
  integer, intent(in) :: i1, i2
  if (i1 /= 0 .and. i2 /= 0) then
    call internal_link_list_append (int%children(i1), i2)
    call internal_link_list_append (int%parents(i2), i1)
  end if
end subroutine interaction_relate

```

Transfer internal parent-child relations defined within interaction `int1` to a new interaction `int` where the particle indices are mapped to. Some particles in `int1` may have no image in `int`. In that case, a child entry maps to zero, and we skip this relation.

Also transfer resonance flags.

*<Interactions: public>+≡*

```
public :: interaction_transfer_relations
```

*<Interactions: procedures>+≡*

```

subroutine interaction_transfer_relations (int1, int2, map)
  type(interaction_t), intent(in) :: int1
  type(interaction_t), intent(inout), target :: int2
  integer, dimension(:), intent(in) :: map
  type(internal_link_t), pointer :: link
  integer :: i, k
  do i = 1, size (map)
    link => internal_link_list_get_first_ptr (int1%parents(i))
    do while (associated (link))
      k = internal_link_get_index (link)

```

```

        call interaction_relate (int2, map(k), map(i))
        call internal_link_advance (link)
    end do
    if (map(i) /= 0) then
        int2%resonant(map(i)) = int1%resonant(i)
    end if
end do
end subroutine interaction_transfer_relations

```

Make up internal parent-child relations for the particle(s) that are connected to a new interaction `int`.

If `resonant` is defined and true, the connections are marked as resonant in the result interaction

```

<Interactions: public>+≡
    public :: interaction_relate_connections

<Interactions: procedures>+≡
    subroutine interaction_relate_connections &
        (int, int_in, connection_index, &
         map, map_connections, resonant)
        type(interaction_t), intent(inout), target :: int
        type(interaction_t), intent(in) :: int_in
        integer, dimension(:), intent(in) :: connection_index
        integer, dimension(:), intent(in) :: map, map_connections
        logical, intent(in), optional :: resonant
        logical :: reson
        integer :: i, i2, k2
        type(internal_link_t), pointer :: link
        reson = .false.; if (present (resonant)) reson = resonant
        do i = 1, size (map_connections)
            k2 = connection_index(i)
            link => internal_link_list_get_first_ptr (int_in%children(k2))
            do while (associated (link))
                i2 = internal_link_get_index (link)
                call interaction_relate (int, map_connections(i), map(i2))
                call internal_link_advance (link)
            end do
            int%resonant(map_connections(i)) = reson
        end do
    end subroutine interaction_relate_connections

```

Return the number of source/target links of the internal connections of particle `i`.

```

<Interactions: public>+≡
    public :: interaction_get_n_children
    public :: interaction_get_n_parents

<Interactions: procedures>+≡
    function interaction_get_n_children (int, i) result (n)
        integer :: n
        type(interaction_t), intent(in) :: int
        integer, intent(in) :: i
        n = internal_link_list_get_length (int%children(i))
    end function interaction_get_n_children

```

```

end function interaction_get_n_children

function interaction_get_n_parents (int, i) result (n)
  integer :: n
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  n = internal_link_list_get_length (int%parents(i))
end function interaction_get_n_parents

```

Return the source/target links of the internal connections of particle *i* as an array.

*(Interactions: public)+≡*

```

public :: interaction_get_children
public :: interaction_get_parents

```

*(Interactions: procedures)+≡*

```

function interaction_get_children (int, i) result (idx)
  integer, dimension(:), allocatable :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  integer :: k
  type(internal_link_t), pointer :: link
  allocate (idx (internal_link_list_get_length (int%children(i))))
  k = 0
  link => internal_link_list_get_first_ptr (int%children(i))
  do while (associated (link))
    k = k + 1
    idx(k) = internal_link_get_index (link)
    call internal_link_advance (link)
  end do
end function interaction_get_children

function interaction_get_parents (int, i) result (idx)
  integer, dimension(:), allocatable :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  integer :: k
  type(internal_link_t), pointer :: link
  allocate (idx (internal_link_list_get_length (int%parents(i))))
  k = 0
  link => internal_link_list_get_first_ptr (int%parents(i))
  do while (associated (link))
    k = k + 1
    idx(k) = internal_link_get_index (link)
    call internal_link_advance (link)
  end do
end function interaction_get_parents

```

Add a source link from an interaction to a corresponding particle within another interaction. These links affect the propagation of particles: the two linked particles are considered as the same particle, outgoing and incoming.

*(Interactions: public)+≡*

```

public :: interaction_set_source_link

```

```

<Interactions: interfaces>+≡
  interface interaction_set_source_link
    module procedure interaction_set_source_link_int
  end interface

```

```

<Interactions: procedures>+≡
  subroutine interaction_set_source_link_int (int, i, int1, i1)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(interaction_t), intent(in), target :: int1
    integer, intent(in) :: i1
    if (i /= 0) call external_link_set (int%source(i), int1, i1)
  end subroutine interaction_set_source_link_int

```

Reassign links to a new interaction (which is an image of the current interaction).

```

<Interactions: public>+≡
  public :: interaction_reassign_links

<Interactions: procedures>+≡
  subroutine interaction_reassign_links (int, int_src, int_target)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in) :: int_src
    type(interaction_t), intent(in), target :: int_target
    integer :: i
    if (allocated (int%source)) then
      do i = 1, size (int%source)
        call external_link_reassign (int%source(i), int_src, int_target)
      end do
    end if
  end subroutine interaction_reassign_links

```

Since links are one-directional, if we want to follow them backwards we have to scan all possibilities. This procedure returns the index of the particle within `int` which points to the particle `i1` within interaction `int1`. If unsuccessful, return zero.

```

<Interactions: public>+≡
  public :: interaction_find_link

<Interactions: procedures>+≡
  function interaction_find_link (int, int1, i1) result (i)
    integer :: i
    type(interaction_t), intent(in) :: int, int1
    integer, intent(in) :: i1
    type(interaction_t), pointer :: int_tmp
    do i = 1, int%n_tot
      int_tmp => external_link_get_ptr (int%source(i))
      if (int_tmp%tag == int1%tag) then
        if (external_link_get_index (int%source(i)) == i1) return
      end if
    end do
    i = 0
  end function interaction_find_link

```

The inverse: return interaction pointer and index for the ultimate source of *i* within *int*.

```

<Interactions: public>+≡
    public :: interaction_find_source

<Interactions: procedures>+≡
    subroutine interaction_find_source (int, i, int1, i1)
        type(interaction_t), intent(in) :: int
        integer, intent(in) :: i
        type(interaction_t), intent(out), pointer :: int1
        integer, intent(out) :: i1
        type(external_link_t) :: link
        link = interaction_get_ultimate_source (int, i)
        int1 => external_link_get_ptr (link)
        i1 = external_link_get_index (link)
    end subroutine interaction_find_source

```

Follow source links recursively to return the ultimate source of a particle.

```

<Interactions: procedures>+≡
    function interaction_get_ultimate_source (int, i) result (link)
        type(external_link_t) :: link
        type(interaction_t), intent(in) :: int
        integer, intent(in) :: i
        type(interaction_t), pointer :: int_src
        integer :: i_src
        link = int%source(i)
        if (external_link_is_set (link)) then
            do
                int_src => external_link_get_ptr (link)
                i_src = external_link_get_index (link)
                if (external_link_is_set (int_src%source(i_src))) then
                    link = int_src%source(i_src)
                else
                    exit
                end if
            end do
        end if
    end function interaction_get_ultimate_source

```

Update mask entries by merging them with corresponding masks in interactions linked to the current one. The mask determines quantum numbers which are summed over.

Note that both the mask of the current interaction and the mask of the linked interaction are updated (side effect!). This ensures that both agree for the linked particle.

```

<Interactions: public>+≡
    public :: interaction_exchange_mask

<Interactions: procedures>+≡
    subroutine interaction_exchange_mask (int)
        type(interaction_t), intent(inout) :: int
        integer :: i, index_link
        type(interaction_t), pointer :: int_link

```

```

do i = 1, int%n_tot
  if (external_link_is_set (int%source(i))) then
    int_link => external_link_get_ptr (int%source(i))
    index_link = external_link_get_index (int%source(i))
    call interaction_merge_mask_entry &
      (int, i, int_link%mask(index_link))
    call interaction_merge_mask_entry &
      (int_link, index_link, int%mask(i))
  end if
end do
call interaction_freeze (int)
end subroutine interaction_exchange_mask

```

Copy momenta from interactions linked to the current one.

```

<Interactions: public>+≡
public :: interaction_receive_momenta

<Interactions: procedures>+≡
subroutine interaction_receive_momenta (int)
  type(interaction_t), intent(inout) :: int
  integer :: i, index_link
  type(interaction_t), pointer :: int_link
  do i = 1, int%n_tot
    if (external_link_is_set (int%source(i))) then
      int_link => external_link_get_ptr (int%source(i))
      index_link = external_link_get_index (int%source(i))
      call interaction_set_momentum (int, int_link%p(index_link), i)
    end if
  end do
end subroutine interaction_receive_momenta

```

The inverse operation: Copy momenta back to the interactions linked to the current one.

```

<Interactions: public>+≡
public :: interaction_send_momenta

<Interactions: procedures>+≡
subroutine interaction_send_momenta (int)
  type(interaction_t), intent(in) :: int
  integer :: i, index_link
  type(interaction_t), pointer :: int_link
  do i = 1, int%n_tot
    if (external_link_is_set (int%source(i))) then
      int_link => external_link_get_ptr (int%source(i))
      index_link = external_link_get_index (int%source(i))
      call interaction_set_momentum (int_link, int%p(i), index_link)
    end if
  end do
end subroutine interaction_send_momenta

```

For numerical comparisons: pacify all momenta in an interaction.

```

<Interactions: public>+≡
public :: interaction_pacify_momenta

```

```

<Interactions: procedures>+≡
  subroutine interaction_pacify_momenta (int, acc)
    type(interaction_t), intent(inout) :: int
    real(default), intent(in) :: acc
    integer :: i
    do i = 1, int%n_tot
      call pacify (int%p(i), acc)
    end do
  end subroutine interaction_pacify_momenta

```

### 7.6.8 Recovering connections

When creating an evaluator for two interactions, we have to know by which particles they are connected. The connection indices can be determined if we have two linked interactions. We assume that `int1` is the source and `int2` the target, so the connections of interest are stored within `int2`. A connection is found if either the source is `int1`, or the (ultimate) source of a particle within `int2` coincides with the (ultimate) source of a particle within `int1`. The result is an array of index pairs.

To make things simple, we scan the interaction twice, once for counting hits, then allocate the array, then scan again and store the connections.

The connections are scanned for `int2`, which has sources in `int1`. It may happen that the order of connections is interchanged (crossed). We require the indices in `int1` to be sorted, so we reorder both index arrays correspondingly before returning them. (After this, the indices in `int2` may be out of order.)

```

<Interactions: public>+≡
  public :: find_connections

<Interactions: procedures>+≡
  subroutine find_connections (int1, int2, n, connection_index)
    type(interaction_t), intent(in) :: int1, int2
    integer, intent(out) :: n
    integer, dimension(:,:), intent(out), allocatable :: connection_index
    integer, dimension(:,:), allocatable :: conn_index_tmp
    integer, dimension(:), allocatable :: ordering
    integer :: i, j, k
    type(external_link_t) :: link2, link1
    type(interaction_t), pointer :: int_link, int_link1
    n = 0
    do i = 1, size (int2%source)
      link2 = interaction_get_ultimate_source (int2, i)
      if (external_link_is_set (link2)) then
        int_link => external_link_get_ptr (link2)
        if (int_link%tag == int1%tag) then
          n = n + 1
        else
          k = external_link_get_index (link2)
          do j = 1, size (int1%source)
            link1 = interaction_get_ultimate_source (int1, j)
            if (external_link_is_set (link1)) then
              int_link1 => external_link_get_ptr (link1)
              if (int_link1%tag == int_link%tag) then
                if (external_link_get_index (link1) == k) then

```

```

                                n = n + 1
                                end if
                                end if
                                end if
                                end do
                                end if
                                end if
                                end do
                                end if
                                end if
                                end do
                                allocate (conn_index_tmp (n, 2))
                                n = 0
                                do i = 1, size (int2%source)
                                    link2 = interaction_get_ultimate_source (int2, i)
                                    if (external_link_is_set (link2)) then
                                        int_link => external_link_get_ptr (link2)
                                        if (int_link%tag == int1%tag) then
                                            n = n + 1
                                            conn_index_tmp(n,1) = external_link_get_index (int2%source(i))
                                            conn_index_tmp(n,2) = i
                                        else
                                            k = external_link_get_index (link2)
                                            do j = 1, size (int1%source)
                                                link1 = interaction_get_ultimate_source (int1, j)
                                                if (external_link_is_set (link1)) then
                                                    int_link1 => external_link_get_ptr (link1)
                                                    if (int_link1%tag == int_link%tag) then
                                                        if (external_link_get_index (link1) == k) then
                                                            n = n + 1
                                                            conn_index_tmp(n,1) = j
                                                            conn_index_tmp(n,2) = i
                                                        end if
                                                    end if
                                                end if
                                            end do
                                        end if
                                    end if
                                end do
                                end if
                                end if
                                end do
                                allocate (connection_index (n, 2))
                                if (n > 1) then
                                    allocate (ordering (n))
                                    ordering = order (conn_index_tmp(:,1))
                                    connection_index = conn_index_tmp(ordering,:)
                                else
                                    connection_index = conn_index_tmp
                                end if
                                end subroutine find_connections

```

### 7.6.9 Test

Generate an interaction of a polarized virtual photon and a colored quark which may be either up or down. Remove the quark polarization. Generate another interaction for the quark radiating a photon and link this to the first interaction. The radiation ignores polarization; transfer this information to the first inter-



action to simplify it. Then, transfer the momentum to the radiating quark and perform a splitting.

*<Interactions: public>+≡*

public :: interaction\_test

*<Interactions: procedures>+≡*

```
subroutine interaction_test ()
  type(interaction_t), target :: int, rad
  type(vector4_t), dimension(3) :: p
  type(quantum_numbers_mask_t), dimension(3) :: mask
  p(2) = vector4_moving (500._default, 500._default, 1)
  p(3) = vector4_moving (500._default, -500._default, 1)
  p(1) = p(2) + p(3)
  call interaction_init (int, 1, 0, 2, set_relations=.true., store_values = .true. )
  call int_set (int, 1, -1, 1, 1, cmplx (0.3_default, 0.1_default, kind=default))
  call int_set (int, 1, -1, -1, 1, cmplx (0.5_default, -0.7_default, kind=default))
  call int_set (int, 1, 1, 1, 1, cmplx (0.1_default, 0._default, kind=default))
  call int_set (int, -1, 1, -1, 2, cmplx (0.4_default, -0.1_default, kind=default))
  call int_set (int, 1, 1, 1, 2, cmplx (0.2_default, 0._default, kind=default))
  call interaction_freeze (int)
  call interaction_set_momenta (int, p)
  mask = new_quantum_numbers_mask (.false., .false., (/ .true., .true., .true. /))
  call interaction_init (rad, 1, 0, 2, mask=mask, set_relations=.true., store_values = .true.)
  call rad_set (1)
  call rad_set (2)
  call interaction_set_source_link (rad, 1, int, 2)
  call interaction_exchange_mask (rad)
  call interaction_receive_momenta (rad)
  p(1) = interaction_get_momentum (rad, 1)
  p(2) = 0.4_default * p(1)
  p(3) = p(1) - p(2)
  call interaction_set_momenta (rad, p(2:3), outgoing=.true.)
  call interaction_freeze (int)
  call interaction_freeze (rad)
  call interaction_set_matrix_element (rad, cmplx (0._default, 0._default, kind=default))
  call interaction_write (int)
  print *
  call interaction_write (rad)
  call interaction_final (int)
  call interaction_final (rad)
contains
  subroutine int_set (int, h1, h2, hq, q, val)
    type(interaction_t), target, intent(inout) :: int
    integer, intent(in) :: h1, h2, hq, q
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    complex(default), intent(in) :: val
    call flavor_init (flv, (/21, q, -q/))
    call color_init_col_acl (col(2), 5, 0)
    call color_init_col_acl (col(3), 0, 5)
    call helicity_init (hel, (/h1, hq, -hq/), (/h2, hq, -hq/))
    call quantum_numbers_init (qn, flv, col, hel)
```

```

        call interaction_add_state (int, qn)
        call interaction_set_matrix_element (int, val)
    end subroutine int_set
    subroutine rad_set (q)
        integer, intent(in) :: q
        type(flavor_t), dimension(3) :: flv
        type(quantum_numbers_t), dimension(3) :: qn
        call flavor_init (flv, (/ q, q, 21 /))
        call quantum_numbers_init (qn, flv)
        call interaction_add_state (rad, qn)
    end subroutine rad_set
end subroutine interaction_test

```

## 7.7 Matrix element evaluation

The `evaluator_t` type is an extension of the `interaction_t` type. It represents either a density matrix as the square of a transition matrix element, or the product of two density matrices. Usually, some quantum numbers are summed over in the result.

The `interaction_t` subobject represents a multi-particle interaction with incoming, virtual, and outgoing particles and the associated (not necessarily diagonal) density matrix of quantum state. When the evaluator is initialized, this interaction is constructed from the input interaction(s).

In addition, the initialization process sets up a multiplication table. For each matrix element of the result, it states which matrix elements are to be taken from the input interaction(s), multiplied (optionally, with an additional weight factor) and summed over.

Eventually, to a processes we associate a chain of evaluators which are to be evaluated sequentially. The physical event and its matrix element value(s) can be extracted from the last evaluator in such a chain.

Evaluators are constructed only once (as long as this is possible) during an initialization step. Then, for each event, momenta are computed and transferred among evaluators using the links within the interaction subobject. The multiplication tables enable fast evaluation of the result without looking at quantum numbers anymore.

`<evaluators.f90>`≡

*<File header>*

`module evaluators`

*<Use kinds>*

*<Use strings>*

*<Use file utils>*

`use diagnostics !NODEP!`

`use lorentz !NODEP!`

`use models`

`use flavors`

`use colors`

`use helicities`

`use quantum_numbers`

```

    use state_matrices
    use interactions

    <Standard module head>

    <Evaluators: public>

    <Evaluators: parameters>

    <Evaluators: types>

    <Evaluators: interfaces>

    contains

    <Evaluators: procedures>

    end module evaluators

```

### 7.7.1 Array of pairings

The evaluator contains an array of `pairing_array` objects. This makes up the multiplication table.

Each pairing array contains two list of matrix element indices and a list of numerical factors. The matrix element indices correspond to the input interactions. The corresponding matrix elements are to be multiplied and optionally multiplied by a factor. The results are summed over to yield one specific matrix element of the result evaluator.

```

<Evaluators: types>≡
    type :: pairing_array_t
        integer, dimension(:), allocatable :: i1, i2
        complex(default), dimension(:), allocatable :: factor
    end type pairing_array_t

<Evaluators: procedures>≡
    elemental subroutine pairing_array_init (pa, n, has_i2, has_factor)
        type(pairing_array_t), intent(out) :: pa
        integer, intent(in) :: n
        logical, intent(in) :: has_i2, has_factor
        allocate (pa%i1 (n))
        if (has_i2) allocate (pa%i2 (n))
        if (has_factor) allocate (pa%factor (n))
    end subroutine pairing_array_init

```

### 7.7.2 The evaluator type

Possible variants of evaluators:

```

<Evaluators: parameters>≡
    integer, parameter :: &
        EVAL_UNDEFINED = 0, &

```

```

EVAL_PRODUCT = 1, &
EVAL_SQUARED_FLOWS = 2, &
EVAL_SQUARE_WITH_COLOR_FACTORS = 3, &
EVAL_COLOR_CONTRACTION = 4, &
EVAL_IDENTITY = 5, &
EVAL_QN_SUM = 6

```

The evaluator type contains the result interaction and an array of pairing lists, one for each matrix element in the result interaction.

```

(Evaluators: public)≡
  public :: evaluator_t

(Evaluators: types)+≡
  type :: evaluator_t
  private
  integer :: type = EVAL_UNDEFINED
  type(interaction_t), pointer :: int_in1 => null ()
  type(interaction_t), pointer :: int_in2 => null ()
  type(interaction_t) :: int
  type(pairing_array_t), dimension(:), allocatable :: pairing_array
contains
  (Evaluators: evaluator: TBP)
end type evaluator_t

```

Output.

```

(Evaluators: public)+≡
  public :: evaluator_write

(Evaluators: evaluator: TBP)≡
  procedure :: write => evaluator_write

(Evaluators: procedures)+≡
  subroutine evaluator_write (eval, unit, &
    verbose, show_momentum_sum, show_mass, show_state, show_table)
    class(evaluator_t), intent(in) :: eval
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
    logical, intent(in), optional :: show_state, show_table
    logical :: conjugate, square, show_tab
    integer :: u, i, j
    u = output_unit (unit); if (u < 0) return
    show_tab = .true.; if (present (show_table)) show_tab = .false.
!   write (u, "(1x,A)") "Evaluator:"
    call interaction_write &
      (eval%int, unit, verbose, show_momentum_sum, show_mass, show_state)
    if (show_tab) then
      write (u, "(1x,A)") "Matrix-element multiplication"
      write (u, "(2x,A)", advance="no") "Input interaction 1:"
      if (associated (eval%int_in1)) then
        write (u, "(1x,I0)") interaction_get_tag (eval%int_in1)
      else
        write (u, "(A)") " [undefined]"
      end if
      write (u, "(2x,A)", advance="no") "Input interaction 2:"
      if (associated (eval%int_in2)) then

```

```

        write (u, "(1x,I0)") interaction_get_tag (eval%int_in2)
    else
        write (u, "(A)") " [undefined]"
    end if
    select case (eval%type)
    case (EVAL_SQUARED_FLOWS, EVAL_SQUARE_WITH_COLOR_FACTORS)
        conjugate = .true.
        square = .true.
    case default
        conjugate = .false.
        square = .false.
    end select
    if (eval%type == EVAL_IDENTITY) then
        write (u, "(1X,A)") "Identity evaluator, pairing array unused"
        return
    end if
    if (allocated (eval%pairing_array)) then
        do i = 1, size (eval%pairing_array)
            write (u, "(2x,A,I0,A)") "ME(", i, ") = "
            do j = 1, size (eval%pairing_array(i)%i1)
                write (u, "(4x,A)", advance="no") "+"
                if (allocated (eval%pairing_array(i)%i2)) then
                    write (u, "(1x,A,I0,A)", advance="no") &
                        "ME1(", eval%pairing_array(i)%i1(j), ")"
                    if (conjugate) then
                        write (u, "(A)", advance="no") "*" x"
                    else
                        write (u, "(A)", advance="no") " x"
                    end if
                    write (u, "(1x,A,I0,A)", advance="no") &
                        "ME2(", eval%pairing_array(i)%i2(j), ")"
                else if (square) then
                    write (u, "(1x,A)", advance="no") "|"
                    write (u, "(A,I0,A)", advance="no") &
                        "ME1(", eval%pairing_array(i)%i1(j), ")"
                    write (u, "(A)", advance="no") "|^2"
                else
                    write (u, "(1x,A,I0,A)", advance="no") &
                        "ME1(", eval%pairing_array(i)%i1(j), ")"
                end if
                if (allocated (eval%pairing_array(i)%factor)) then
                    write (u, "(1x,A)", advance="no") "x"
                    write (u, "(1x,'(',')ES19.12,'(',')ES19.12,')')") &
                        eval%pairing_array(i)%factor(j)
                else
                    write (u, *)
                end if
            end do
        end do
    end if
    ! print *, size (eval%pairing_array)
end if
end subroutine evaluator_write

```

Assignment: Deep copy of the interaction component.

```

<Evaluators: public>+=
  public :: assignment(=)

<Evaluators: interfaces>≡
  interface assignment(=)
    module procedure evaluator_assign
  end interface

<Evaluators: procedures>+=
  subroutine evaluator_assign (eval_out, eval_in)
    type(evaluator_t), intent(out) :: eval_out
    type(evaluator_t), intent(in) :: eval_in
    eval_out%type = eval_in%type
    eval_out%int_in1 => eval_in%int_in1
    eval_out%int_in2 => eval_in%int_in2
    eval_out%int = eval_in%int
    if (allocated (eval_in%pairing_array)) then
      allocate (eval_out%pairing_array (size (eval_in%pairing_array)))
      eval_out%pairing_array = eval_in%pairing_array
    end if
  end subroutine evaluator_assign

```

Replace the interactions. The dipoles use this to avoid error prone code duplication. This is only meaningful if the interactions are copies of the original ones, otherwise, the result will most likely put the program to a horrible death.

```

<Evaluators: public>+=
  public :: evaluator_replace_interaction

<Evaluators: procedures>+=
  subroutine evaluator_replace_interaction (eval, int1, int2)
    type(evaluator_t), intent(inout) :: eval
    type(interaction_t), intent(in), target, optional :: int1, int2
    if (eval%type == EVAL_UNDEFINED) return
    if (present (int1)) then
      call interaction_reassign_links (eval%int, eval%int_in1, int1)
      eval%int_in1 => int1
    end if
    if (present (int2)) then
      call interaction_reassign_links (eval%int, eval%int_in2, int2)
      eval%int_in2 => int2
    end if
  end subroutine evaluator_replace_interaction

```

### 7.7.3 Auxiliary structures for evaluator creation

Creating an evaluator that properly handles all quantum numbers requires some bookkeeping. In this section, we define several auxiliary types and methods that organize and simplify this task. More structures are defined within the specific initializers (as local types and internal subroutines).

These types are currently implemented in a partial object-oriented way: We define some basic methods for initialization etc. here, but the evaluator routines

below do access their internals as well. This simplifies some things such as index addressing using array slices, at the expense of losing some clarity.

## Index mapping

Index mapping are abundant when constructing an evaluator. To have arrays of index mappings, we define this:

```

(Evaluators: types)+≡
  type :: index_map_t
    integer, dimension(:), allocatable :: entry
  end type index_map_t

(Evaluators: procedures)+≡
  elemental subroutine index_map_init (map, n)
    type(index_map_t), intent(out) :: map
    integer, intent(in) :: n
    allocate (map%entry (n))
  end subroutine index_map_init

(Evaluators: procedures)+≡
  function index_map_exists (map) result (flag)
    logical :: flag
    type(index_map_t), intent(in) :: map
    flag = allocated (map%entry)
  end function index_map_exists

(Evaluators: interfaces)+≡
  interface size
    module procedure index_map_size
  end interface

(Evaluators: procedures)+≡
  function index_map_size (map) result (s)
    integer :: s
    type(index_map_t), intent(in) :: map
    if (allocated (map%entry)) then
      s = size (map%entry)
    else
      s = 0
    end if
  end function index_map_size

(Evaluators: interfaces)+≡
  interface assignment(=)
    module procedure index_map_assign_int
    module procedure index_map_assign_array
  end interface

(Evaluators: procedures)+≡
  elemental subroutine index_map_assign_int (map, ival)
    type(index_map_t), intent(inout) :: map

```

```

    integer, intent(in) :: ival
    map%entry = ival
end subroutine index_map_assign_int

subroutine index_map_assign_array (map, array)
    type(index_map_t), intent(inout) :: map
    integer, dimension(:), intent(in) :: array
    map%entry = array
end subroutine index_map_assign_array

```

*(Evaluators: procedures)*+≡

```

elemental subroutine index_map_set_entry (map, i, ival)
    type(index_map_t), intent(inout) :: map
    integer, intent(in) :: i
    integer, intent(in) :: ival
    map%entry(i) = ival
end subroutine index_map_set_entry

```

*(Evaluators: procedures)*+≡

```

elemental function index_map_get_entry (map, i) result (ival)
    integer :: ival
    type(index_map_t), intent(in) :: map
    integer, intent(in) :: i
    ival = map%entry(i)
end function index_map_get_entry

```

## Index mapping (two-dimensional)

This is a variant with a square matrix instead of an array.

*(Evaluators: types)*+≡

```

type :: index_map2_t
    integer :: s = 0
    integer, dimension(:,:), allocatable :: entry
end type index_map2_t

```

*(Evaluators: procedures)*+≡

```

elemental subroutine index_map2_init (map, n)
    type(index_map2_t), intent(out) :: map
    integer, intent(in) :: n
    map%s = n
    allocate (map%entry (n, n))
end subroutine index_map2_init

```

*(Evaluators: procedures)*+≡

```

function index_map2_exists (map) result (flag)
    logical :: flag
    type(index_map2_t), intent(in) :: map
    flag = allocated (map%entry)
end function index_map2_exists

```



```

<Evaluators: interfaces>+≡
interface size
  module procedure index_map2_size
end interface

<Evaluators: procedures>+≡
function index_map2_size (map) result (s)
  integer :: s
  type(index_map2_t), intent(in) :: map
  s = map%s
end function index_map2_size

<Evaluators: interfaces>+≡
interface assignment(=)
  module procedure index_map2_assign_int
end interface

<Evaluators: procedures>+≡
elemental subroutine index_map2_assign_int (map, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: ival
  map%entry = ival
end subroutine index_map2_assign_int

<Evaluators: procedures>+≡
elemental subroutine index_map2_set_entry (map, i, j, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: i, j
  integer, intent(in) :: ival
  map%entry(i,j) = ival
end subroutine index_map2_set_entry

<Evaluators: procedures>+≡
elemental function index_map2_get_entry (map, i, j) result (ival)
  integer :: ival
  type(index_map2_t), intent(in) :: map
  integer, intent(in) :: i, j
  ival = map%entry(i,j)
end function index_map2_get_entry

```

### Auxiliary structures: particle mask

This is a simple container of a logical array.

```

<Evaluators: types>+≡
type :: prt_mask_t
  logical, dimension(:), allocatable :: entry
end type prt_mask_t

```

```

(Evaluators: procedures)+≡
subroutine prt_mask_init (mask, n)
  type(prt_mask_t), intent(out) :: mask
  integer, intent(in) :: n
  allocate (mask%entry (n))
end subroutine prt_mask_init

```

```

(Evaluators: interfaces)+≡
interface size
  module procedure prt_mask_size
end interface

```

```

(Evaluators: procedures)+≡
function prt_mask_size (mask) result (s)
  integer :: s
  type(prt_mask_t), intent(in) :: mask
  s = size (mask%entry)
end function prt_mask_size

```

## Quantum number containers

Trivial transparent containers:

```

(Evaluators: types)+≡
type :: qn_list_t
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
end type qn_list_t

type :: qn_mask_array_t
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
end type qn_mask_array_t

```

## Auxiliary structures: connection entries

This type is used as intermediate storage when computing the product of two evaluators or the square of an evaluator. The quantum-number array **qn** corresponds to the particles common to both interactions, but irrelevant quantum numbers (color) masked out. The index arrays **index\_in** determine the entries in the input interactions that contribute to this connection. **n\_index** is the size of these arrays, and **count** is used while filling the entries. Finally, the quantum-number arrays **qn\_in\_list** are the actual entries in the input interaction that contribute. In the product case, they exclude the connected quantum numbers.

Each evaluator has its own **connection\_table** which contains an array of **connection\_entry** objects, but also has stuff that specifically applies to the evaluator type. Hence, we do not generalize the **connection\_table\_t** type.

The filling procedure **connection\_entry\_add\_state** is specific to the various evaluator types.

```

(Evaluators: types)+≡
type :: connection_entry_t
  type(quantum_numbers_t), dimension(:), allocatable :: qn_conn

```

```

integer, dimension(:), allocatable :: n_index
integer, dimension(:), allocatable :: count
type(index_map_t), dimension(:), allocatable :: index_in
type(qn_list_t), dimension(:), allocatable :: qn_in_list
end type connection_entry_t

```

(Evaluators: procedures)+≡

```

subroutine connection_entry_init &
  (entry, n_count, n_map, qn_conn, count, n_rest)
  type(connection_entry_t), intent(out) :: entry
  integer, intent(in) :: n_count, n_map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_conn
  integer, dimension(n_count), intent(in) :: count
  integer, dimension(n_count), intent(in) :: n_rest
  integer :: i
  allocate (entry%qn_conn (size (qn_conn)))
  allocate (entry%n_index (n_count))
  allocate (entry%count (n_count))
  allocate (entry%index_in (n_map))
  allocate (entry%qn_in_list (n_count))
  entry%qn_conn = qn_conn
  entry%n_index = count
  entry%count = 0
  if (size (entry%index_in) == size (count)) then
    call index_map_init (entry%index_in, count)
  else
    call index_map_init (entry%index_in, count(1))
  end if
  do i = 1, n_count
    allocate (entry%qn_in_list(i)%qn (n_rest(i), count(i)))
  end do
end subroutine connection_entry_init

```

(Evaluators: procedures)+≡

```

subroutine connection_entry_write (entry, unit)
  type(connection_entry_t), intent(in) :: entry
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  call quantum_numbers_write (entry%qn_conn, unit)
  write (u, *)
  do i = 1, size (entry%n_index)
    write (u, *) "Input interaction", i
    do j = 1, entry%n_index(i)
      if (size (entry%n_index) == size (entry%index_in)) then
        write (u, "(2x,I0,4x,I0,2x)", advance = "no") &
          j, index_map_get_entry (entry%index_in(i), j)
      else
        write (u, "(2x,I0,4x,I0,2x,I0,2x)", advance = "no") &
          j, index_map_get_entry (entry%index_in(1), j), &
            index_map_get_entry (entry%index_in(2), j)
      end if
    end do
  end do
end subroutine connection_entry_write

```

```

        call quantum_numbers_write (entry%qn_in_list(i)%qn(:,j), unit)
        write (u, *)
    end do
end do
end subroutine connection_entry_write

```

## Color handling

For managing color-factor computation, we introduce this local type. The `index` is the index in the color table that corresponds to a given matrix element index in the input interaction. The `col` array stores the color assignments in rows. The `factor` array associates a complex number with each pair of arrays in the color table. The `factor_is_known` array reveals whether a given factor is known already or still has to be computed.

*(Evaluators: types)+≡*

```

type :: color_table_t
    integer, dimension(:), allocatable :: index
    type(color_t), dimension(:, :), allocatable :: col
    logical, dimension(:, :), allocatable :: factor_is_known
    complex(default), dimension(:, :), allocatable :: factor
end type color_table_t

```

This is the initializer. We extract the color states from the given state matrices, establish index mappings between the two states (implemented by the array `me_index`), make an array of color states, and initialize the color-factor table. The latter is two-dimensional (includes interference) and not yet filled.

*(Evaluators: procedures)+≡*

```

subroutine color_table_init (color_table, state, n_tot)
    type(color_table_t), intent(out) :: color_table
    type(state_matrix_t), intent(in) :: state
    integer, intent(in) :: n_tot
    type(state_iterator_t) :: it
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    type(state_matrix_t) :: state_col
    integer :: index, n_col_state
    allocate (color_table%index &
        (state_matrix_get_n_matrix_elements (state)))
    color_table%index = 0
    allocate (qn (n_tot))
    call state_matrix_init (state_col)
    call state_iterator_init (it, state)
    do while (state_iterator_is_valid (it))
        index = state_iterator_get_me_index (it)
        call quantum_numbers_init (qn, col = state_iterator_get_color (it))
        call state_matrix_add_state (state_col, qn, &
            me_index = color_table%index(index))
        call state_iterator_advance (it)
    end do
    n_col_state = state_matrix_get_n_matrix_elements (state_col)
    allocate (color_table%col (n_tot, n_col_state))
    call state_iterator_init (it, state_col)

```

```

do while (state_iterator_is_valid (it))
  index = state_iterator_get_me_index (it)
  color_table%col(:,index) = state_iterator_get_color (it)
  call state_iterator_advance (it)
end do
call state_matrix_final (state_col)
allocate (color_table%factor_is_known (n_col_state, n_col_state))
allocate (color_table%factor (n_col_state, n_col_state))
color_table%factor_is_known = .false.
end subroutine color_table_init

```

Output (debugging use):

*(Evaluators: procedures)+≡*

```

subroutine color_table_write (color_table, unit)
  type(color_table_t), intent(in) :: color_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  write (u, *) "Color table:"
  if (allocated (color_table%index)) then
    write (u, *) "  Index mapping state => color table:"
    do i = 1, size (color_table%index)
      write (u, "(3x,I0,2x,I0,2x)" i, color_table%index(i)
    end do
    write (u, *) "  Color table:"
    do i = 1, size (color_table%col, 2)
      write (u, "(3x,I0,2x)", advance = "no") i
      call color_write (color_table%col(:,i), unit)
      write (u, *)
    end do
    write (u, *) "  Defined color factors:"
    do i = 1, size (color_table%factor, 1)
      do j = 1, size (color_table%factor, 2)
        if (color_table%factor_is_known(i,j)) then
          write (u, *) i, j, color_table%factor(i,j)
        end if
      end do
    end do
  end if
end subroutine color_table_write

```

This subroutine sets color factors, based on information from the hard matrix element: the list of pairs of color-flow indices (in the basis of the matrix element code), the list of corresponding factors, and the list of mappings from the matrix element index in the input interaction to the color-flow index in the hard matrix element object.

We first determine the mapping of color-flow indices from the hard matrix element code to the current color table. The mapping could be nontrivial because the latter is derived from iterating over a state matrix, which may return states in non-canonical order. The translation table can be determined because we have, for the complete state matrix, both the mapping to the hard interac-

tion (the input `col_index_hi`) and the mapping to the current color table (the component `color_table%index`).

Once this mapping is known, we scan the list of index pairs `color_flow_index` and translate them to valid color-table index pairs. For this pair, the color factor is set using the corresponding entry in the list `col_factor`.

*(Evaluators: procedures)+≡*

```
subroutine color_table_set_color_factors (color_table, &
    col_flow_index, col_factor, col_index_hi)
    type(color_table_t), intent(inout) :: color_table
    integer, dimension(:,:), intent(in) :: col_flow_index
    complex(default), dimension(:), intent(in) :: col_factor
    integer, dimension(:), intent(in) :: col_index_hi
    integer, dimension(:), allocatable :: hi_to_ct
    integer :: n_cflow
    integer :: hi_index, me_index, ct_index, cf_index
    integer, dimension(2) :: hi_index_pair, ct_index_pair
    n_cflow = size (col_index_hi)
    if (size (color_table%index) /= n_cflow) &
        call msg_bug ("Mismatch between hard matrix element and color table")
    allocate (hi_to_ct (n_cflow))
    do me_index = 1, size (color_table%index)
        ct_index = color_table%index(me_index)
        hi_index = col_index_hi(me_index)
        hi_to_ct(hi_index) = ct_index
    end do
    do cf_index = 1, size (col_flow_index, 2)
        hi_index_pair = col_flow_index(:,cf_index)
        ct_index_pair = hi_to_ct(hi_index_pair)
        color_table%factor(ct_index_pair(1), ct_index_pair(2)) = &
            col_factor(cf_index)
        color_table%factor_is_known(ct_index_pair(1), ct_index_pair(2)) = .true.
    end do
end subroutine color_table_set_color_factors
```

This function returns a color factor, given two indices which point to the matrix elements of the initial state matrix. Internally, we can map them to the corresponding indices in the color table. As a side effect, we store the color factor in the color table for later lookup. (I.e., this function is impure.)

*(Evaluators: procedures)+≡*

```
function color_table_get_color_factor (color_table, index1, index2, nc) &
    result (factor)
    real(default) :: factor
    type(color_table_t), intent(inout) :: color_table
    integer, intent(in) :: index1, index2
    integer, intent(in), optional :: nc
    integer :: i1, i2
!   print *, "compute color factor ", index1, index2
    i1 = color_table%index(index1)
    i2 = color_table%index(index2)
!   print *, " indices = ", i1, i2
    if (color_table%factor_is_known(i1,i2)) then
        factor = color_table%factor(i1,i2)
!   print *, " is known : ", factor
```

```

else
    factor = compute_color_factor &
        (color_table%col(:,i1), color_table%col(:,i2), nc)
    color_table%factor(i1,i2) = factor
    color_table%factor_is_known(i1,i2) = .true.
!    print *, "    computed : ", factor
end if
end function color_table_get_color_factor

```

#### 7.7.4 Creating an evaluator: Matrix multiplication

The evaluator for matrix multiplication is the most complicated variant.

The initializer takes two input interactions and constructs the result evaluator, which consists of the interaction and the multiplication table for the product (or convolution) of the two. Normally, the input interactions are connected by one or more common particles (e.g., decay, structure function convolution).

In the result interaction, quantum numbers of the connections can be summed over. This is determined by the `qn_mask_conn` argument. The `qn_mask_rest` argument is its analog for the other particles within the result interaction. (E.g., for the trace of the state matrix, all quantum numbers are summed over.) Finally, the `connections_are_resonant` argument tells whether the connecting particles should be marked as resonant in the final event record. This is useful for decays.

The algorithm consists of the following steps:

1. **find\_connections**: Find the particles which are connected, i.e., common to both input interactions. Either they are directly linked, or both are linked to a common source.
2. **compute\_index\_bounds\_and\_mappings**: Compute the mappings of particle indices from the input interactions to the result interaction. There is a separate mapping for the connected particles.
3. **accumulate\_connected\_states**: Create an auxiliary state matrix which lists the possible quantum numbers for the connected particles. When building this matrix, count the number of times each assignment is contained in any of the input states and, for each of the input states, record the index of the matrix element within the new state matrix. For the connected particles, reassign color indices such that no color state is present twice in different color-index assignment. Note that helicity assignments of the connected state can be (and will be) off-diagonal, so no spin correlations are lost in decays.

Do this for both input interactions.

4. **allocate\_connection\_entries**: Allocate a table of connections. Each table row corresponds to one state in the auxiliary matrix, and to multiple states of the input interactions. It collects all states of the unconnected particles in the two input interactions that are associated with the particular state (quantum-number assignment) of the connected particles.

5. **fill\_connection\_table**: Fill the table of connections by scanning both input interactions. When copying states, reassign color indices for the unconnected particles such that they match between all involved particle sets (interaction 1, interaction 2, and connected particles).
6. **make\_product\_interaction**: Scan the table of connections we have just built. For each entry, construct all possible pairs of states of the unconnected particles and combine them with the specific connected-particle state. This is a possible quantum-number assignment of the result interaction. Now mask all quantum numbers that should be summed over, and append this to the result state matrix. Record the matrix element index of the result. We now have the result interaction.
7. **make\_pairing\_array**: First allocate the pairing array with the number of entries of the result interaction. Then scan the table of connections again. For each entry, record the indices of the matrix elements which have to be multiplied and summed over in order to compute this particular matrix element. This makes up the multiplication table.
8. **record\_links**: Transfer all source pointers from the input interactions to the result interaction. Do the same for the internal parent-child relations and resonance assignments. For the connected particles, make up appropriate additional parent-child relations. This allows for fetching momenta from other interactions when a new event is filled, and to reconstruct the event history when the event is analyzed.

After all this is done, for each event, we just have to evaluate the pairing arrays (multiplication tables) in order to compute the result matrix elements in their proper positions. The quantum-number assignments remain fixed from now on.

```

(Evaluators: public)+≡
    public :: evaluator_init_product

(Evaluators: interfaces)+≡
    interface evaluator_init_product
        module procedure evaluator_init_product_ii
        module procedure evaluator_init_product_ie
        module procedure evaluator_init_product_ei
        module procedure evaluator_init_product_ee
    end interface

(Evaluators: procedures)+≡
    subroutine evaluator_init_product_ii &
        (eval, int_in1, int_in2, qn_mask_conn, qn_mask_rest, &
         connections_are_resonant)

        type(evaluator_t), intent(out), target :: eval
        type(interaction_t), intent(in), target :: int_in1, int_in2
        type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
        type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
        logical, intent(in), optional :: connections_are_resonant

        type(qn_mask_array_t), dimension(2) :: qn_mask_in

```



```

type :: connection_table_t
  integer :: n_conn = 0
  integer, dimension(2) :: n_rest = 0
  integer :: n_tot = 0
  integer :: n_me_conn = 0
  type(state_matrix_t) :: state
  type(index_map_t), dimension(:), allocatable :: index_conn
  type(connection_entry_t), dimension(:), allocatable :: entry
  type(index_map_t) :: index_result
end type connection_table_t
type(connection_table_t) :: connection_table

integer :: n_in, n_vir, n_out, n_tot
integer, dimension(2) :: n_rest
integer :: n_conn, n_me_conn

integer, dimension(:,:), allocatable :: connection_index
type(index_map_t), dimension(2) :: prt_map_in
type(index_map_t) :: prt_map_conn
type(prt_mask_t), dimension(2) :: prt_is_connected
type(quantum_numbers_mask_t), dimension(:), allocatable :: &
  qn_mask_conn_initial

integer :: i

eval%type = EVAL_PRODUCT
eval%int_in1 => int_in1
eval%int_in2 => int_in2
!   print *, "Evaluator product"
!   print *, "First interaction"
!   call interaction_write (int_in1)
!   print *
!   print *, "Second interaction"
!   call interaction_write (int_in2)
!   print *

call find_connections (int_in1, int_in2, n_conn, connection_index)
if (n_conn == 0) then
  call msg_message ("First interaction:")
  call interaction_write (int_in1)
  call msg_message ("Second interaction:")
  call interaction_write (int_in2)
  call msg_fatal ("Evaluator product: no connections found between factors")
end if
call compute_index_bounds_and_mappings &
  (int_in1, int_in2, n_conn, &
   n_in, n_vir, n_out, n_tot, &
   n_rest, prt_map_in, prt_map_conn)

call prt_mask_init (prt_is_connected(1), interaction_get_n_tot (int_in1))
call prt_mask_init (prt_is_connected(2), interaction_get_n_tot (int_in2))
do i = 1, 2
  prt_is_connected(i)%entry = .true.

```

```

        prt_is_connected(i)%entry(connection_index(:,i)) = .false.
    end do
    allocate (qn_mask_conn_initial (n_conn))
    qn_mask_conn_initial = &
        interaction_get_mask (int_in1, connection_index(:,1)) .or. &
        interaction_get_mask (int_in2, connection_index(:,2))
    allocate (qn_mask_in(1)%mask (interaction_get_n_tot (int_in1)))
    allocate (qn_mask_in(2)%mask (interaction_get_n_tot (int_in2)))
    qn_mask_in(1)%mask = interaction_get_mask (int_in1)
    qn_mask_in(2)%mask = interaction_get_mask (int_in2)

    call connection_table_init (connection_table, &
        interaction_get_state_matrix_ptr (int_in1), &
        interaction_get_state_matrix_ptr (int_in2), &
        qn_mask_conn_initial, &
        n_conn, connection_index, n_rest)
    call connection_table_fill (connection_table, &
        interaction_get_state_matrix_ptr (int_in1), &
        interaction_get_state_matrix_ptr (int_in2), &
        connection_index, prt_is_connected)
    call make_product_interaction (eval%int, &
        n_in, n_vir, n_out, &
        connection_table, &
        prt_map_in, prt_is_connected, &
        qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
!    call connection_table_write (connection_table)
    call make_pairing_array (eval%pairing_array, &
        interaction_get_n_matrix_elements (eval%int), &
        connection_table)
    call record_links (eval%int, &
        int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
        prt_is_connected, connections_are_resonant)
    call connection_table_final (connection_table)

!    print *, "Result evaluator"
!    call evaluator_write (eval)

    if (interaction_get_n_matrix_elements (eval%int) == 0) then
        print *, "Evaluator product"
        print *, "First interaction"
        call interaction_write (int_in1)
        print *
        print *, "Second interaction"
        call interaction_write (int_in2)
        print *
        call msg_fatal ("Product of density matrices is empty", &
            (/ var_str (" -----"), &
            var_str ("This happens when two density matrices are convoluted "), &
            var_str ("but the processes they belong to (e.g., production "), &
            var_str ("and decay) do not match. This could happen if the "), &
            var_str ("beam specification does not match the hard "), &
            var_str ("process. Or it may indicate a WHIZARD bug.") /) )
    end if

```

contains

```

subroutine compute_index_bounds_and_mappings &
  (int1, int2, n_conn, &
   n_in, n_vir, n_out, n_tot, &
   n_rest, prt_map_in, prt_map_conn)
  type(interaction_t), intent(in) :: int1, int2
  integer, intent(in) :: n_conn
  integer, intent(out) :: n_in, n_vir, n_out, n_tot
  integer, dimension(2), intent(out) :: n_rest
  type(index_map_t), dimension(2), intent(out) :: prt_map_in
  type(index_map_t), intent(out) :: prt_map_conn
  integer, dimension(:), allocatable :: index
  integer :: n_in1, n_vir1, n_out1
  integer :: n_in2, n_vir2, n_out2
  integer :: k
  n_in1 = interaction_get_n_in (int1)
  n_vir1 = interaction_get_n_vir (int1)
  n_out1 = interaction_get_n_out (int1) - n_conn
  n_rest(1) = n_in1 + n_vir1 + n_out1
  n_in2 = interaction_get_n_in (int2) - n_conn
  n_vir2 = interaction_get_n_vir (int2)
  n_out2 = interaction_get_n_out (int2)
  n_rest(2) = n_in2 + n_vir2 + n_out2
  n_in = n_in1 + n_in2
  n_vir = n_vir1 + n_vir2 + n_conn
  n_out = n_out1 + n_out2
  n_tot = n_in + n_vir + n_out
  call index_map_init (prt_map_in, n_rest)
  call index_map_init (prt_map_conn, n_conn)
  allocate (index (n_tot))
  index = (/ (i, i = 1, n_tot) /)
  prt_map_in(1)%entry(1 : n_in1) = index( 1 : n_in1)
  k = n_in1
  prt_map_in(2)%entry(1 : n_in2) = index(k+1 : k+n_in2)
  k = k + n_in2
  prt_map_in(1)%entry(n_in1+1 : n_in1+n_vir1) = index(k+1 : k+n_vir1)
  k = k + n_vir1
  prt_map_in(2)%entry(n_in2+1 : n_in2+n_vir2) = index(k+1 : k+n_vir2)
  k = k + n_vir2
  prt_map_conn%entry = index(k+1 : k+n_conn)
  k = k + n_conn
  prt_map_in(1)%entry(n_in1+n_vir1+1 : n_rest(1)) = index(k+1 : k+n_out1)
  k = k + n_out1
  prt_map_in(2)%entry(n_in2+n_vir2+1 : n_rest(2)) = index(k+1 : k+n_out2)
end subroutine compute_index_bounds_and_mappings

subroutine connection_table_init &
  (connection_table, state_in1, state_in2, qn_mask_conn, &
   n_conn, connection_index, n_rest)
  type(connection_table_t), intent(out) :: connection_table
  type(state_matrix_t), intent(in), target :: state_in1, state_in2
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_conn
  integer, intent(in) :: n_conn

```

```

integer, dimension(:,:), intent(in) :: connection_index
integer, dimension(2), intent(in) :: n_rest
integer, dimension(2) :: n_me_in
type(state_iterator_t) :: it
type(quantum_numbers_t), dimension(n_conn) :: qn
integer :: i, me_index_in, me_index_conn, n_me_conn
integer, dimension(2) :: me_count
connection_table%n_conn = n_conn
connection_table%n_rest = n_rest
n_me_in(1) = state_matrix_get_n_matrix_elements (state_in1)
n_me_in(2) = state_matrix_get_n_matrix_elements (state_in2)
allocate (connection_table%index_conn (2))
call index_map_init (connection_table%index_conn, n_me_in)
connection_table%index_conn = 0
call state_matrix_init (connection_table%state, n_counters=2)
do i = 1, 2
  select case (i)
    case (1); call state_iterator_init (it, state_in1)
    case (2); call state_iterator_init (it, state_in2)
  end select
  do while (state_iterator_is_valid (it))
    qn = state_iterator_get_quantum_numbers (it, connection_index(:,i))
    call quantum_numbers_undefine (qn, qn_mask_conn)
    call quantum_numbers_canonicalize_color (qn)
    me_index_in = state_iterator_get_me_index (it)
    call state_matrix_add_state (connection_table%state, qn, &
      counter_index = i, me_index = me_index_conn)
    call index_map_set_entry (connection_table%index_conn(i), &
      me_index_in, me_index_conn)
    call state_iterator_advance (it)
  end do
end do
n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
connection_table%n_me_conn = n_me_conn
allocate (connection_table%entry (n_me_conn))
call state_iterator_init (it, connection_table%state)
do while (state_iterator_is_valid (it))
  i = state_iterator_get_me_index (it)
  me_count = state_iterator_get_me_count (it)
  call connection_entry_init (connection_table%entry(i), 2, 2, &
    state_iterator_get_quantum_numbers (it), &
    me_count, n_rest)
  call state_iterator_advance (it)
end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
  type(connection_table_t), intent(inout) :: connection_table
  call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
  type(connection_table_t), intent(in) :: connection_table
  integer, intent(in), optional :: unit

```

```

integer :: i, j
integer :: u
u = output_unit (unit)
write (u, *) "Connection table:"
call state_matrix_write (connection_table%state, unit)
if (allocated (connection_table%index_conn)) then
  write (u, *) "  Index mapping input => connection table:"
  do i = 1, size (connection_table%index_conn)
    write (u, *) "    Input state", i
    do j = 1, size (connection_table%index_conn(i))
      write (u, *)      j, &
        index_map_get_entry (connection_table%index_conn(i), j)
    end do
  end do
end if
if (allocated (connection_table%entry)) then
  write (u, *) "  Connection table contents:"
  do i = 1, size (connection_table%entry)
    call connection_entry_write (connection_table%entry(i), unit)
  end do
end if
if (index_map_exists (connection_table%index_result)) then
  write (u, *) "  Index mapping connection table => output:"
  do i = 1, size (connection_table%index_result)
    write (u, *)      i, &
      index_map_get_entry (connection_table%index_result, i)
  end do
end if
end subroutine connection_table_write

subroutine connection_table_fill &
  (connection_table, state_in1, state_in2, &
   connection_index, prt_is_connected)
type(connection_table_t), intent(inout) :: connection_table
type(state_matrix_t), intent(in), target :: state_in1, state_in2
integer, dimension(:, :), intent(in) :: connection_index
type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
type(state_iterator_t) :: it
integer :: index_in, index_conn
integer :: color_offset
integer :: n_result_entries
integer :: i, k
color_offset = state_matrix_get_max_color_value (connection_table%state)
do i = 1, 2
  select case (i)
    case (1); call state_iterator_init (it, state_in1)
    case (2); call state_iterator_init (it, state_in2)
  end select
  do while (state_iterator_is_valid (it))
    index_in = state_iterator_get_me_index (it)
    index_conn = index_map_get_entry &
      (connection_table%index_conn(i), index_in)
    if (index_conn /= 0) then
      call connection_entry_add_state &

```

```

        (connection_table%entry(index_conn), i, &
         index_in, &
         state_iterator_get_quantum_numbers (it), &
         connection_index(:,i), prt_is_connected(i), &
         color_offset)
    end if
    call state_iterator_advance (it)
end do
color_offset = color_offset &
+ state_matrix_get_max_color_value (state_in1)
end do
n_result_entries = 0
do k = 1, size (connection_table%entry)
    n_result_entries = &
        n_result_entries + product (connection_table%entry(k)%n_index)
end do
call index_map_init (connection_table%index_result, n_result_entries)
end subroutine connection_table_fill

subroutine connection_entry_add_state &
    (entry, i, index_in, qn_in, connection_index, prt_is_connected, &
     color_offset)
type(connection_entry_t), intent(inout) :: entry
integer, intent(in) :: i
integer, intent(in) :: index_in
type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
integer, dimension(:), intent(in) :: connection_index
type(prt_mask_t), intent(in) :: prt_is_connected
integer, intent(in) :: color_offset
integer :: c, k
integer, dimension(:,:), allocatable :: color_map
entry%count(i) = entry%count(i) + 1
c = entry%count(i)
call quantum_numbers_set_color_map &
    (color_map, qn_in(connection_index), entry%qn_conn)
call index_map_set_entry (entry%index_in(i), c, index_in)
entry%qn_in_list(i)%qn(:,c) = pack (qn_in, prt_is_connected%entry)
call quantum_numbers_translate_color &
    (entry%qn_in_list(i)%qn(:,c), color_map, color_offset)
end subroutine connection_entry_add_state

subroutine make_product_interaction (int, &
    n_in, n_vir, n_out, &
    connection_table, &
    prt_map_in, prt_is_connected, &
    qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
type(interaction_t), intent(out), target :: int
integer, intent(in) :: n_in, n_vir, n_out
type(connection_table_t), intent(inout), target :: connection_table
type(index_map_t), dimension(2), intent(in) :: prt_map_in
type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
type(qn_mask_array_t), dimension(2), intent(in) :: qn_mask_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: &
    qn_mask_conn_initial

```

```

type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
type(index_map_t), dimension(2) :: prt_index_in
type(index_map_t) :: prt_index_conn
integer :: n_tot, n_conn
integer, dimension(2) :: n_rest
integer :: i, j, k, m
type(quantum_numbers_t), dimension(:), allocatable :: qn
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
type(connection_entry_t), pointer :: entry
integer :: result_index
n_conn = connection_table%n_conn
n_rest = connection_table%n_rest
n_tot = sum (n_rest) + n_conn
allocate (qn (n_tot), qn_mask (n_tot))
do i = 1, 2
    call index_map_init (prt_index_in(i), n_rest(i))
    prt_index_in(i) = &
        prt_map_in(i)%entry ((/ (j, j = 1, n_rest(i)) /))
end do
call index_map_init (prt_index_conn, n_conn)
prt_index_conn = prt_map_conn%entry ((/ (j, j = 1, n_conn) /))
do i = 1, 2
    if (present (qn_mask_rest)) then
        qn_mask(prt_index_in(i)%entry) = &
            pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry) &
            .or. qn_mask_rest
    else
        qn_mask(prt_index_in(i)%entry) = &
            pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry)
    end if
end do
qn_mask(prt_index_conn%entry) = qn_mask_conn_initial .or. qn_mask_conn
call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
m = 1
do i = 1, connection_table%n_me_conn
    entry => connection_table%entry(i)
    qn(prt_index_conn%entry) = &
        quantum_numbers_undefined (entry%qn_conn, qn_mask_conn)
    do j = 1, entry%n_index(1)
        qn(prt_index_in(1)%entry) = entry%qn_in_list(1)%qn(:,j)
        do k = 1, entry%n_index(2)
            qn(prt_index_in(2)%entry) = entry%qn_in_list(2)%qn(:,k)
            call interaction_add_state (int, qn, me_index = result_index)
            call index_map_set_entry &
                (connection_table%index_result, m, result_index)
            m = m + 1
        end do
    end do
end do
call interaction_freeze (int)
end subroutine make_product_interaction

subroutine make_pairing_array (pa, n_matrix_elements, connection_table)

```

```

type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
integer, intent(in) :: n_matrix_elements
type(connection_table_t), intent(in), target :: connection_table
type(connection_entry_t), pointer :: entry
integer, dimension(:), allocatable :: n_entries
integer :: i, j, k, m, r
allocate (pa (n_matrix_elements))
allocate (n_entries (n_matrix_elements))
n_entries = 0
do m = 1, size (connection_table%index_result)
    r = index_map_get_entry (connection_table%index_result, m)
    n_entries(r) = n_entries(r) + 1
end do
call pairing_array_init &
    (pa, n_entries, has_i2=.true., has_factor=.false.)
m = 1
n_entries = 0
do i = 1, connection_table%n_me_conn
    entry => connection_table%entry(i)
    do j = 1, entry%n_index(1)
        do k = 1, entry%n_index(2)
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            pa(r)%i1(n_entries(r)) = &
                index_map_get_entry (entry%index_in(1), j)
            pa(r)%i2(n_entries(r)) = &
                index_map_get_entry (entry%index_in(2), k)
            m = m + 1
        end do
    end do
end do
end subroutine make_pairing_array

subroutine record_links (int, &
    int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
    prt_is_connected, connections_are_resonant)
type(interaction_t), intent(inout) :: int
type(interaction_t), intent(in), target :: int_in1, int_in2
integer, dimension(:,:), intent(in) :: connection_index
type(index_map_t), dimension(2), intent(in) :: prt_map_in
type(index_map_t), intent(in) :: prt_map_conn
type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
logical, intent(in), optional :: connections_are_resonant
type(index_map_t), dimension(2) :: prt_map_all
integer :: i, j, k, ival
call index_map_init (prt_map_all(1), size (prt_is_connected(1)))
k = 0
j = 0
do i = 1, size (prt_is_connected(1))
    if (prt_is_connected(1)%entry(i)) then
        j = j + 1
        ival = index_map_get_entry (prt_map_in(1), j)
        call index_map_set_entry (prt_map_all(1), i, ival)
    else

```



```

        k = k + 1
        ival = index_map_get_entry (prt_map_conn, k)
        call index_map_set_entry (prt_map_all(1), i, ival)
    end if
    call interaction_set_source_link (int, ival, int_in1, i)
end do
call interaction_transfer_relations (int_in1, int, prt_map_all(1)%entry)
call index_map_init (prt_map_all(2), size (prt_is_connected(2)))
j = 0
do i = 1, size (prt_is_connected(2))
    if (prt_is_connected(2)%entry(i)) then
        j = j + 1
        ival = index_map_get_entry (prt_map_in(2), j)
        call index_map_set_entry (prt_map_all(2), i, ival)
        call interaction_set_source_link (int, ival, int_in2, i)
    else
        call index_map_set_entry (prt_map_all(2), i, 0)
    end if
end do
call interaction_transfer_relations (int_in2, int, prt_map_all(2)%entry)
call interaction_relate_connections (int, &
    int_in2, connection_index(:,2), prt_map_all(2)%entry, &
    prt_map_conn%entry, connections_are_resonant)
end subroutine record_links

end subroutine evaluator_init_product_ii

subroutine evaluator_init_product_ie &
    (eval, int_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
    connections_are_resonant)
type(evaluator_t), intent(out), target :: eval
type(interaction_t), intent(in), target :: int_in1
type(evaluator_t), intent(in), target :: eval_in2
type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
logical, intent(in), optional :: connections_are_resonant
call evaluator_init_product_ii &
    (eval, int_in1, eval_in2%int, qn_mask_conn, qn_mask_rest, &
    connections_are_resonant)
end subroutine evaluator_init_product_ie

subroutine evaluator_init_product_ei &
    (eval, eval_in1, int_in2, qn_mask_conn, qn_mask_rest, &
    connections_are_resonant)
type(evaluator_t), intent(out), target :: eval
type(evaluator_t), intent(in), target :: eval_in1
type(interaction_t), intent(in), target :: int_in2
type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
logical, intent(in), optional :: connections_are_resonant
call evaluator_init_product_ii &
    (eval, eval_in1%int, int_in2, qn_mask_conn, qn_mask_rest, &
    connections_are_resonant)
end subroutine evaluator_init_product_ei

```

```

subroutine evaluator_init_product_ee &
  (eval, eval_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
   connections_are_resonant)
  type(evaluator_t), intent(out), target :: eval
  type(evaluator_t), intent(in), target :: eval_in1, eval_in2
  type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
  type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
  logical, intent(in), optional :: connections_are_resonant
  call evaluator_init_product_ii &
    (eval, eval_in1%int, eval_in2%int, qn_mask_conn, qn_mask_rest, &
     connections_are_resonant)
end subroutine evaluator_init_product_ee

```

### 7.7.5 Creating an evaluator: square

The generic initializer for an evaluator that squares a matrix element. Depending on the provided mask, we select the appropriate specific initializer for either diagonal or non-diagonal helicity density matrices.

```

(Evaluators: public)+≡
  public :: evaluator_init_square

(Evaluators: procedures)+≡
  subroutine evaluator_init_square (eval, int_in, qn_mask, &
    col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
    type(evaluator_t), intent(out), target :: eval
    type(interaction_t), intent(in), target :: int_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, dimension(:,:), intent(in), optional :: col_flow_index
    complex(default), dimension(:), intent(in), optional :: col_factor
    integer, dimension(:), intent(in), optional :: col_index_hi
    logical, intent(in), optional :: expand_color_flows
    integer, intent(in), optional :: nc
    if (all (quantum_numbers_mask_diagonal_helicity (qn_mask))) then
      call evaluator_init_square_diag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
    else
      call evaluator_init_square_nondiag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
    end if
  end subroutine evaluator_init_square

```

#### Color-summed squared matrix (diagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. The mask must be such that off-diagonal matrix elements are excluded.

If `color_flows` is set, the evaluator keeps color-flow entries separate and drops all interfering color structures. The color factors are set to unity in this case.

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic

purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to multiplication, with a few notable differences:

1. The connected particles are known, the correspondence is one-to-one. All particles are connected, and the mapping of indices is trivial, which simplifies the following steps.
2. **accumulate\_connected\_states**: The matrix of connected states encompasses all particles, but color indices are removed. However, ghost states are still kept separate from physical color states. No color-index reassignment is necessary.
3. The table of connections contains single index and quantum-number arrays instead of pairs of them. They are paired with themselves in all possible ways.
4. **make\_squared\_interaction**: Now apply the predefined quantum-numbers mask, which usually collects all color states (physical and ghosts), and possibly a helicity sum.
5. **make\_pairing\_array**: For each pair of input states, compute the color factor (including a potential ghost-parity sign) and store this in the pairing array together with the matrix-element indices for multiplication.
6. **record\_links**: This is again trivial due to the one-to-one correspondence.

*(Evaluators: public)*+≡

```
public :: evaluator_init_square_diag
```

*(Evaluators: procedures)*+≡

```
subroutine evaluator_init_square_diag (eval, int_in, qn_mask, &
    col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
```

```

type(evaluator_t), intent(out), target :: eval
type(interaction_t), intent(in), target :: int_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
integer, dimension(:,:), intent(in), optional :: col_flow_index
complex(default), dimension(:), intent(in), optional :: col_factor
integer, dimension(:), intent(in), optional :: col_index_hi
logical, intent(in), optional :: expand_color_flows
integer, intent(in), optional :: nc

integer :: n_in, n_vir, n_out, n_tot
integer :: n_me_in
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial

type :: connection_table_t
integer :: n_tot = 0
integer :: n_me_conn = 0
type(state_matrix_t) :: state
type(index_map_t) :: index_conn
type(connection_entry_t), dimension(:), allocatable :: entry
type(index_map_t) :: index_result
end type connection_table_t
```

```

type(connection_table_t) :: connection_table

logical :: sum_colors
type(color_table_t) :: color_table

if (present (expand_color_flows)) then
    sum_colors = .not. expand_color_flows
else
    sum_colors = .true.
end if

if (sum_colors) then
    eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
else
    eval%type = EVAL_SQUARED_FLOWS
end if
eval%int_in1 => int_in

!     print *, "Interaction square with color factors (diag)"
!     print *, "Input interaction"
!     call interaction_write (int_in)

n_in = interaction_get_n_in (int_in)
n_vir = interaction_get_n_vir (int_in)
n_out = interaction_get_n_out (int_in)
n_tot = interaction_get_n_tot (int_in)

allocate (qn_mask_initial (n_tot))
qn_mask_initial = interaction_get_mask (int_in)
call quantum_numbers_mask_set_color &
    (qn_mask_initial, sum_colors, mask_cg=.false.)
if (sum_colors) then
    call color_table_init &
        (color_table, interaction_get_state_matrix_ptr (int_in), n_tot)
    if (present (col_flow_index) .and. present (col_factor) &
        .and. present (col_index_hi)) then
        call color_table_set_color_factors &
            (color_table, col_flow_index, col_factor, col_index_hi)
    end if
!     call color_table_write (color_table)
end if

call connection_table_init (connection_table, &
    interaction_get_state_matrix_ptr (int_in), &
    qn_mask_initial, qn_mask, n_tot)
call connection_table_fill (connection_table, &
    interaction_get_state_matrix_ptr (int_in))
call make_squared_interaction (eval%int, &
    n_in, n_vir, n_out, n_tot, &
    connection_table, sum_colors, qn_mask_initial .or. qn_mask)
call make_pairing_array (eval%pairing_array, &
    interaction_get_n_matrix_elements (eval%int), &
    connection_table, sum_colors, color_table, n_in, n_tot, nc)
call record_links (eval%int, int_in, n_tot)

```

```

    call connection_table_final (connection_table)
!    print *, "Result evaluator:"
!    call evaluator_write (eval)

contains

subroutine connection_table_init &
    (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
    type(connection_table_t), intent(out) :: connection_table
    type(state_matrix_t), intent(in), target :: state_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, intent(in) :: n_tot
    type(quantum_numbers_t), dimension(n_tot) :: qn
    type(state_iterator_t) :: it
    integer :: i, n_me_in, me_index_in
    integer :: me_index_conn, n_me_conn
    integer, dimension(1) :: me_count
    connection_table%n_tot = n_tot
    n_me_in = state_matrix_get_n_matrix_elements (state_in)
    call index_map_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call state_matrix_init (connection_table%state, n_counters=1)
    call state_iterator_init (it, state_in)
    do while (state_iterator_is_valid (it))
        qn = state_iterator_get_quantum_numbers (it)
        if (all (quantum_numbers_are_physical (qn, qn_mask))) then
            call quantum_numbers_undefine (qn, qn_mask_in)
            me_index_in = state_iterator_get_me_index (it)
            call state_matrix_add_state (connection_table%state, qn, &
                counter_index = 1, me_index = me_index_conn)
            call index_map_set_entry (connection_table%index_conn, &
                me_index_in, me_index_conn)
        end if
        call state_iterator_advance (it)
    end do
    n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
    connection_table%n_me_conn = n_me_conn
    allocate (connection_table%entry (n_me_conn))
    call state_iterator_init (it, connection_table%state)
    do while (state_iterator_is_valid (it))
        i = state_iterator_get_me_index (it)
        me_count = state_iterator_get_me_count (it)
        call connection_entry_init (connection_table%entry(i), 1, 2, &
            state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
        call state_iterator_advance (it)
    end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
    type(connection_table_t), intent(inout) :: connection_table
    call state_matrix_final (connection_table%state)
end subroutine connection_table_final

```

```

subroutine connection_table_write (connection_table, unit)
  type(connection_table_t), intent(in) :: connection_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  write (u, *) "Connection table:"
  call state_matrix_write (connection_table%state, unit)
  if (index_map_exists (connection_table%index_conn)) then
    write (u, *) "  Index mapping input => connection table:"
    do i = 1, size (connection_table%index_conn)
      write (u, *) i, &
        index_map_get_entry (connection_table%index_conn, i)
    end do
  end if
  if (allocated (connection_table%entry)) then
    write (u, *) "  Connection table contents"
    do i = 1, size (connection_table%entry)
      call connection_entry_write (connection_table%entry(i), unit)
    end do
  end if
  if (index_map_exists (connection_table%index_result)) then
    write (u, *) "  Index mapping connection table => output"
    do i = 1, size (connection_table%index_result)
      write (u, *) i, &
        index_map_get_entry (connection_table%index_result, i)
    end do
  end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
  type(connection_table_t), intent(inout) :: connection_table
  type(state_matrix_t), intent(in), target :: state
  integer :: index_in, index_conn, n_result_entries
  type(state_iterator_t) :: it
  integer :: k
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
    index_in = state_iterator_get_me_index (it)
    index_conn = &
      index_map_get_entry (connection_table%index_conn, index_in)
    if (index_conn /= 0) then
      call connection_entry_add_state &
        (connection_table%entry(index_conn), &
          index_in, &
          state_iterator_get_quantum_numbers (it))
    end if
    call state_iterator_advance (it)
  end do
  n_result_entries = 0
  do k = 1, size (connection_table%entry)
    n_result_entries = &
      n_result_entries + connection_table%entry(k)%n_index(1) ** 2
  end do

```

```

        call index_map_init (connection_table%index_result, n_result_entries)
        connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index_in, qn_in)
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: index_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer :: c
    entry%count = entry%count + 1
    c = entry%count(1)
    call index_map_set_entry (entry%index_in(1), c, index_in)
    entry%qn_in_list(1)%qn(:,c) = qn_in
end subroutine connection_entry_add_state

subroutine make_squared_interaction (int, &
    n_in, n_vir, n_out, n_tot, &
    connection_table, sum_colors, qn_mask)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out, n_tot
    type(connection_table_t), intent(inout), target :: connection_table
    logical, intent(in) :: sum_colors
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index, n_contrib
    integer :: i, m
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 0
    do i = 1, connection_table%n_me_conn
        entry => connection_table%entry(i)
        qn = quantum_numbers_undefined (entry%qn_conn, qn_mask)
        if (.not. sum_colors) call quantum_numbers_invert_color (qn(1:n_in))
        call interaction_add_state (int, qn, me_index = result_index)
        n_contrib = entry%n_index(1) ** 2
        connection_table%index_result%entry(m+1:m+n_contrib) = result_index
        m = m + n_contrib
    end do
    call interaction_freeze (int)
end subroutine make_squared_interaction

subroutine make_pairing_array (pa, &
    n_matrix_elements, connection_table, sum_colors, color_table, &
    n_in, n_tot, nc)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
    logical, intent(in) :: sum_colors
    type(color_table_t), intent(inout) :: color_table
    type(connection_entry_t), pointer :: entry
    integer, intent(in) :: n_in, n_tot
    integer, intent(in), optional :: nc
    integer, dimension(:), allocatable :: n_entries
    integer :: i, j, k, l, ks, ls, m, r

```

```

integer :: color_multiplicity_in
allocate (pa (n_matrix_elements))
allocate (n_entries (n_matrix_elements))
n_entries = 0
do m = 1, size (connection_table%index_result)
    r = index_map_get_entry (connection_table%index_result, m)
    n_entries(r) = n_entries(r) + 1
end do
call pairing_array_init &
    (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
m = 1
n_entries = 0
do i = 1, connection_table%n_me_conn
    entry => connection_table%entry(i)
    do k = 1, entry%n_index(1)
        if (sum_colors) then
            color_multiplicity_in = &
                product (abs (quantum_numbers_get_color_type &
                    (entry%qn_in_list(1)%qn(:n_in, k))))
            do l = 1, entry%n_index(1)
                r = index_map_get_entry (connection_table%index_result, m)
                n_entries(r) = n_entries(r) + 1
                ks = index_map_get_entry (entry%index_in(1), k)
                ls = index_map_get_entry (entry%index_in(1), l)
                pa(r)%i1(n_entries(r)) = ks
                pa(r)%i2(n_entries(r)) = ls
                pa(r)%factor(n_entries(r)) = &
                    color_table_get_color_factor (color_table, ks, ls, nc) &
                    / color_multiplicity_in
                m = m + 1
            end do
        else
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            ks = index_map_get_entry (entry%index_in(1), k)
            pa(r)%i1(n_entries(r)) = ks
            m = m + 1
        end if
    end do
end do
end subroutine make_pairing_array

subroutine record_links (int, int_in, n_tot)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in), target :: int_in
    integer, intent(in) :: n_tot
    integer, dimension(n_tot) :: map
    integer :: i
    do i = 1, n_tot
        call interaction_set_source_link (int, i, int_in, i)
    end do
    map = (/ (i, i = 1, n_tot) /)
    call interaction_transfer_relations (int_in, int, map)
end subroutine record_links

```



```
end subroutine evaluator_init_square_diag
```

### Color-summed squared matrix (support nodiagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. Unless requested otherwise by the quantum-number mask, the result contains off-diagonal matrix elements. (The input interaction must be diagonal since it represents an amplitude, not a density matrix.)

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to the previous one, with some additional complications due to the necessity to loop over two helicity indices.

```
(Evaluators: public)+≡
public :: evaluator_init_square_nondiag

(Evaluators: procedures)+≡
subroutine evaluator_init_square_nondiag (eval, int_in, qn_mask, &
    col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

    type(evaluator_t), intent(out), target :: eval
    type(interaction_t), intent(in), target :: int_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, dimension(:,:), intent(in), optional :: col_flow_index
    complex(default), dimension(:), intent(in), optional :: col_factor
    integer, dimension(:), intent(in), optional :: col_index_hi
    logical, intent(in), optional :: expand_color_flows
    integer, intent(in), optional :: nc

    integer :: n_in, n_vir, n_out, n_tot
    integer :: n_me_in
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial

    type :: connection_table_t
        integer :: n_tot = 0
        integer :: n_me_conn = 0
        type(state_matrix_t) :: state
        type(index_map2_t) :: index_conn
        type(connection_entry_t), dimension(:), allocatable :: entry
        type(index_map_t) :: index_result
    end type connection_table_t
    type(connection_table_t) :: connection_table

    logical :: sum_colors
    type(color_table_t) :: color_table

    if (present (expand_color_flows)) then
        sum_colors = .not. expand_color_flows
    else
        sum_colors = .true.
    end if
end subroutine evaluator_init_square_nondiag
```

```

end if

if (sum_colors) then
    eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
else
    eval%type = EVAL_SQUARED_FLOWS
end if
eval%int_in1 => int_in

!    print *, "Interaction square with color factors (nondiag)"
!    print *, "Input interaction"
!    call interaction_write (int_in)
n_in = interaction_get_n_in (int_in)
n_vir = interaction_get_n_vir (int_in)
n_out = interaction_get_n_out (int_in)
n_tot = interaction_get_n_tot (int_in)

allocate (qn_mask_initial (n_tot))
qn_mask_initial = interaction_get_mask (int_in)
call quantum_numbers_mask_set_color &
    (qn_mask_initial, sum_colors, mask_cg=.false.)
if (sum_colors) then
    call color_table_init &
        (color_table, interaction_get_state_matrix_ptr (int_in), n_tot)
    if (present (col_flow_index) .and. present (col_factor) &
        .and. present (col_index_hi)) then
        call color_table_set_color_factors &
            (color_table, col_flow_index, col_factor, col_index_hi)
    end if
!    call color_table_write (color_table)
end if

call connection_table_init (connection_table, &
    interaction_get_state_matrix_ptr (int_in), &
    qn_mask_initial, qn_mask, n_tot)
call connection_table_fill (connection_table, &
    interaction_get_state_matrix_ptr (int_in))
call make_squared_interaction (eval%int, &
    n_in, n_vir, n_out, n_tot, &
    connection_table, sum_colors, qn_mask_initial .or. qn_mask)
!    call connection_table_write (connection_table)
call make_pairing_array (eval%pairing_array, &
    interaction_get_n_matrix_elements (eval%int), &
    connection_table, sum_colors, color_table, n_in, n_tot, nc)
call record_links (eval%int, int_in, n_tot)
call connection_table_final (connection_table)

!    print *, "Result evaluator:"
!    call evaluator_write (eval)

contains

subroutine connection_table_init &
    (connection_table, state_in, qn_mask_in, qn_mask, n_tot)

```

```

type(connection_table_t), intent(out) :: connection_table
type(state_matrix_t), intent(in), target :: state_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
integer, intent(in) :: n_tot
type(quantum_numbers_t), dimension(n_tot) :: qn1, qn2, qn
type(state_iterator_t) :: it1, it2, it
integer :: i, n_me_in, me_index_in1, me_index_in2
integer :: me_index_conn, n_me_conn
integer, dimension(1) :: me_count
connection_table%n_tot = n_tot
n_me_in = state_matrix_get_n_matrix_elements (state_in)
call index_map2_init (connection_table%index_conn, n_me_in)
connection_table%index_conn = 0
call state_matrix_init (connection_table%state, n_counters=1)
call state_iterator_init (it1, state_in)
do while (state_iterator_is_valid (it1))
  qn1 = state_iterator_get_quantum_numbers (it1)
  me_index_in1 = state_iterator_get_me_index (it1)
  call state_iterator_init (it2, state_in)
  do while (state_iterator_is_valid (it2))
    qn2 = state_iterator_get_quantum_numbers (it2)
    if (all (quantum_numbers_are_compatible (qn1, qn2, qn_mask))) then
      qn = qn1 .merge. qn2
      call quantum_numbers_undefine (qn, qn_mask_in)
      me_index_in2 = state_iterator_get_me_index (it2)
      call state_matrix_add_state (connection_table%state, qn, &
        counter_index = 1, me_index = me_index_conn)
      call index_map2_set_entry (connection_table%index_conn, &
        me_index_in1, me_index_in2, me_index_conn)
    end if
    call state_iterator_advance (it2)
  end do
  call state_iterator_advance (it1)
end do
n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
connection_table%n_me_conn = n_me_conn
allocate (connection_table%entry (n_me_conn))
call state_iterator_init (it, connection_table%state)
do while (state_iterator_is_valid (it))
  i = state_iterator_get_me_index (it)
  me_count = state_iterator_get_me_count (it)
  call connection_entry_init (connection_table%entry(i), 1, 2, &
    state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
  call state_iterator_advance (it)
end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
  type(connection_table_t), intent(inout) :: connection_table
  call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)

```

```

type(connection_table_t), intent(in) :: connection_table
integer, intent(in), optional :: unit
integer :: i, j
integer :: u
u = output_unit (unit)
write (u, *) "Connection table:"
call state_matrix_write (connection_table%state, unit)
if (index_map2_exists (connection_table%index_conn)) then
  write (u, *) " Index mapping input => connection table:"
  do i = 1, size (connection_table%index_conn)
    do j = 1, size (connection_table%index_conn)
      write (u, *) i, j, &
        index_map2_get_entry (connection_table%index_conn, i, j)
    end do
  end do
end if
if (allocated (connection_table%entry)) then
  write (u, *) " Connection table contents"
  do i = 1, size (connection_table%entry)
    call connection_entry_write (connection_table%entry(i), unit)
  end do
end if
if (index_map_exists (connection_table%index_result)) then
  write (u, *) " Index mapping connection table => output"
  do i = 1, size (connection_table%index_result)
    write (u, *) i, &
      index_map_get_entry (connection_table%index_result, i)
  end do
end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
type(connection_table_t), intent(inout), target :: connection_table
type(state_matrix_t), intent(in), target :: state
integer :: index1_in, index2_in, index_conn, n_result_entries
type(state_iterator_t) :: it1, it2
integer :: k
call state_iterator_init (it1, state)
do while (state_iterator_is_valid (it1))
  index1_in = state_iterator_get_me_index (it1)
  call state_iterator_init (it2, state)
  do while (state_iterator_is_valid (it2))
    index2_in = state_iterator_get_me_index (it2)
    index_conn = index_map2_get_entry &
      (connection_table%index_conn, index1_in, index2_in)
    if (index_conn /= 0) then
      call connection_entry_add_state &
        (connection_table%entry(index_conn), &
          index1_in, index2_in, &
          state_iterator_get_quantum_numbers (it1) &
          .merge. &
          state_iterator_get_quantum_numbers (it2))
    end if
    call state_iterator_advance (it2)
  end do
end do

```

```

        end do
        call state_iterator_advance (it1)
    end do
    n_result_entries = 0
    do k = 1, size (connection_table%entry)
        n_result_entries = &
            n_result_entries + connection_table%entry(k)%n_index(1)
    end do
    call index_map_init (connection_table%index_result, n_result_entries)
    connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index1_in, index2_in, qn_in)
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: index1_in, index2_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer :: c
    entry%count = entry%count + 1
    c = entry%count(1)
    call index_map_set_entry (entry%index_in(1), c, index1_in)
    call index_map_set_entry (entry%index_in(2), c, index2_in)
    entry%qn_in_list(1)%qn(:,c) = qn_in
end subroutine connection_entry_add_state

subroutine make_squared_interaction (int, &
    n_in, n_vir, n_out, n_tot, &
    connection_table, sum_colors, qn_mask)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out, n_tot
    type(connection_table_t), intent(inout), target :: connection_table
    logical, intent(in) :: sum_colors
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index
    integer :: i, k, m
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 0
    do i = 1, connection_table%n_me_conn
        entry => connection_table%entry(i)
        do k = 1, size (entry%qn_in_list(1)%qn, 2)
            qn = quantum_numbers_undefined &
                (entry%qn_in_list(1)%qn(:,k), qn_mask)
            if (.not. sum_colors) &
                call quantum_numbers_invert_color (qn(1:n_in))
            call interaction_add_state (int, qn, me_index = result_index)
            call index_map_set_entry (connection_table%index_result, m + 1, &
                result_index)
            m = m + 1
        end do
    end do
    call interaction_freeze (int)
end subroutine make_squared_interaction

```

```

subroutine make_pairing_array (pa, &
    n_matrix_elements, connection_table, sum_colors, color_table, &
    n_in, n_tot, nc)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
    logical, intent(in) :: sum_colors
    type(color_table_t), intent(inout) :: color_table
    type(connection_entry_t), pointer :: entry
    integer, intent(in) :: n_in, n_tot
    integer, intent(in), optional :: nc
    integer, dimension(:), allocatable :: n_entries
    integer :: i, k, k1s, k2s, m, r
    integer :: color_multiplicity_in
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
    n_entries = 0
    do m = 1, size (connection_table%index_result)
        r = index_map_get_entry (connection_table%index_result, m)
        n_entries(r) = n_entries(r) + 1
    end do
    call pairing_array_init &
        (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
    m = 1
    n_entries = 0
    do i = 1, connection_table%n_me_conn
        entry => connection_table%entry(i)
        do k = 1, entry%n_index(1)
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            if (sum_colors) then
                k1s = index_map_get_entry (entry%index_in(1), k)
                k2s = index_map_get_entry (entry%index_in(2), k)
                pa(r)%i1(n_entries(r)) = k1s
                pa(r)%i2(n_entries(r)) = k2s
                color_multiplicity_in = &
                    product (abs (quantum_numbers_get_color_type &
                        (entry%qn_in_list(1)%qn(:n_in, k))))
                pa(r)%factor(n_entries(r)) = &
                    color_table_get_color_factor (color_table, k1s, k2s, nc) &
                    / color_multiplicity_in
            else
                k1s = index_map_get_entry (entry%index_in(1), k)
                pa(r)%i1(n_entries(r)) = k1s
            end if
            m = m + 1
        end do
    end do
end subroutine make_pairing_array

subroutine record_links (int, int_in, n_tot)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in), target :: int_in
    integer, intent(in) :: n_tot

```

```

integer, dimension(n_tot) :: map
integer :: i
do i = 1, n_tot
    call interaction_set_source_link (int, i, int_in, i)
end do
map = (/ (i, i = 1, n_tot) /)
call interaction_transfer_relations (int_in, int, map)
end subroutine record_links

end subroutine evaluator_init_square_nondiag

```

### Copy with additional contracted color states

This evaluator involves no square or multiplication, its matrix elements are just copies of the (single) input interaction. However, the state matrix of the interaction contains additional states that have color indices contracted. This is used for copies of the beam or structure-function interactions that need to match the hard interaction also in the case where its color indices coincide.

*(Evaluators: public)*+≡

```
public :: evaluator_init_color_contractions
```

*(Evaluators: procedures)*+≡

```

subroutine evaluator_init_color_contractions (eval, int_in)
    type(evaluator_t), intent(out), target :: eval
    type(interaction_t), intent(in), target :: int_in
    integer :: n_in, n_vir, n_out, n_tot
    type(state_matrix_t) :: state_with_contractions
    integer, dimension(:), allocatable :: me_index
    integer, dimension(:), allocatable :: result_index
    eval%type = EVAL_COLOR_CONTRACTION
    eval%int_in1 => int_in
!    print *, "Interaction with additional color contractions"
!    print *, "Input interaction"
!    call interaction_write (int_in)
    n_in = interaction_get_n_in (int_in)
    n_vir = interaction_get_n_vir (int_in)
    n_out = interaction_get_n_out (int_in)
    n_tot = interaction_get_n_tot (int_in)
    state_with_contractions = interaction_get_state_matrix_ptr (int_in)
    call state_matrix_add_color_contractions (state_with_contractions)
    call make_contracted_interaction (eval%int, &
        me_index, result_index, &
        n_in, n_vir, n_out, n_tot, &
        state_with_contractions, interaction_get_mask (int_in))
    call make_pairing_array (eval%pairing_array, me_index, result_index)
    call record_links (eval%int, int_in, n_tot)
!    print *, "Result evaluator:"
!    call evaluator_write (eval)

```

contains

```

subroutine make_contracted_interaction (int, &
    me_index, result_index, &

```

```

        n_in, n_vir, n_out, n_tot, state, qn_mask)
type(interaction_t), intent(out), target :: int
integer, dimension(:), intent(out), allocatable :: me_index
integer, dimension(:), intent(out), allocatable :: result_index
integer, intent(in) :: n_in, n_vir, n_out, n_tot
type(state_matrix_t), intent(in) :: state
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
type(state_iterator_t) :: it
integer :: n_me, i
type(quantum_numbers_t), dimension(n_tot) :: qn
call interaction_init (int, n_in, n_vir, n_out, mask=qn_mask)
n_me = state_matrix_get_n_leaves (state)
allocate (me_index (n_me))
allocate (result_index (n_me))
call state_iterator_init (it, state)
i = 0
do while (state_iterator_is_valid (it))
    i = i + 1
    me_index(i) = state_iterator_get_me_index (it)
    qn = state_iterator_get_quantum_numbers (it)
    call interaction_add_state (int, qn, me_index = result_index(i))
    call state_iterator_advance (it)
end do
call interaction_freeze (int)
end subroutine make_contracted_interaction

subroutine make_pairing_array (pa, me_index, result_index)
type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
integer, dimension(:), intent(in) :: me_index, result_index
integer, dimension(:), allocatable :: n_entries
integer :: n_matrix_elements, r, i
n_matrix_elements = size (me_index)
allocate (pa (n_matrix_elements))
allocate (n_entries (n_matrix_elements))
n_entries = 1
call pairing_array_init &
    (pa, n_entries, has_i2=.false., has_factor=.false.)
do i = 1, n_matrix_elements
    r = result_index(i)
    pa(r)%i1(1) = me_index(i)
end do
end subroutine make_pairing_array

subroutine record_links (int, int_in, n_tot)
type(interaction_t), intent(inout) :: int
type(interaction_t), intent(in), target :: int_in
integer, intent(in) :: n_tot
integer, dimension(n_tot) :: map
integer :: i
do i = 1, n_tot
    call interaction_set_source_link (int, i, int_in, i)
end do
map = (/ (i, i = 1, n_tot) /)
call interaction_transfer_relations (int_in, int, map)

```



```

        end subroutine record_links

    end subroutine evaluator_init_color_contractions

```

### Auxiliary procedure for initialization

This will become a standard procedure in F2008. The result is true if the number of true values in the mask is odd. We use the function for determining the ghost parity of a quantum-number array.

```

<Evaluators: procedures>+≡
    function parity (mask)
        logical :: parity
        logical, dimension(:) :: mask
        integer :: i
        parity = .false.
        do i = 1, size (mask)
            if (mask(i)) parity = .not. parity
        end do
    end function parity

```

### 7.7.6 Accessing contents

Return the interaction component, as a pointer to avoid any copying.

```

<Evaluators: public>+≡
    public :: evaluator_get_int_ptr

<Evaluators: procedures>+≡
    function evaluator_get_int_ptr (eval) result (int)
        type(interaction_t), pointer :: int
        type(evaluator_t), intent(in), target :: eval
        int => eval%int
    end function evaluator_get_int_ptr

```

### 7.7.7 Inherited procedures

Return true if the state matrix within the interaction is empty.

```

<Evaluators: public>+≡
    public :: evaluator_is_empty

<Evaluators: procedures>+≡
    function evaluator_is_empty (eval) result (flag)
        logical :: flag
        type(evaluator_t), intent(in) :: eval
        flag = interaction_is_empty (eval%int)
    end function evaluator_is_empty

```

Return the integer tag.

```

<Evaluators: public>+≡
    public :: evaluator_get_tag

```

```

⟨Evaluators: procedures⟩+=
  function evaluator_get_tag (eval) result (tag)
    integer :: tag
    type(evaluator_t), intent(in) :: eval
    tag = interaction_get_tag (eval%int)
  end function evaluator_get_tag

```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

```

⟨Evaluators: public⟩+=
  public :: evaluator_get_norm

⟨Evaluators: procedures⟩+=
  function evaluator_get_norm (eval) result (norm)
    real(default) :: norm
    type(evaluator_t), intent(in) :: eval
    norm = interaction_get_norm (eval%int)
  end function evaluator_get_norm

```

Return the number of particles.

```

⟨Evaluators: public⟩+=
  public :: evaluator_get_n_tot
  public :: evaluator_get_n_in
  public :: evaluator_get_n_vir
  public :: evaluator_get_n_out

⟨Evaluators: procedures⟩+=
  function evaluator_get_n_tot (eval) result (n_tot)
    integer :: n_tot
    type(evaluator_t), intent(in) :: eval
    n_tot = interaction_get_n_tot (eval%int)
  end function evaluator_get_n_tot

  function evaluator_get_n_in (eval) result (n_in)
    integer :: n_in
    type(evaluator_t), intent(in) :: eval
    n_in = interaction_get_n_in (eval%int)
  end function evaluator_get_n_in

  function evaluator_get_n_vir (eval) result (n_vir)
    integer :: n_vir
    type(evaluator_t), intent(in) :: eval
    n_vir = interaction_get_n_vir (eval%int)
  end function evaluator_get_n_vir

  function evaluator_get_n_out (eval) result (n_out)
    integer :: n_out
    type(evaluator_t), intent(in) :: eval
    n_out = interaction_get_n_out (eval%int)
  end function evaluator_get_n_out

```

Sum all matrix element values.

```

⟨Evaluators: public⟩+=
  public :: evaluator_sum

```

```

<Evaluators: procedures>+≡
  function evaluator_sum (eval) result (value)
    complex(default) :: value
    type(evaluator_t), intent(in) :: eval
    value = interaction_sum (eval%int)
  end function evaluator_sum

```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

```

<Evaluators: public>+≡
  public :: evaluator_normalize_by_trace

<Evaluators: procedures>+≡
  subroutine evaluator_normalize_by_trace (eval)
    type(evaluator_t), intent(inout) :: eval
    call interaction_normalize_by_trace (eval%int)
  end subroutine evaluator_normalize_by_trace

```

Analogous, but renormalize by maximal (absolute) value.

```

<Evaluators: public>+≡
  public :: evaluator_normalize_by_max

<Evaluators: procedures>+≡
  subroutine evaluator_normalize_by_max (eval)
    type(evaluator_t), intent(inout) :: eval
    call interaction_normalize_by_max (eval%int)
  end subroutine evaluator_normalize_by_max

```

Append color-contracted states. Matrix element array and multiplication table are unaffected by this.

```

<Evaluators: public>+≡
  public :: evaluator_add_color_contractions

<Evaluators: procedures>+≡
  subroutine evaluator_add_color_contractions (eval)
    type(evaluator_t), intent(inout) :: eval
    call interaction_add_color_contractions (eval%int)
  end subroutine evaluator_add_color_contractions

```

Return the quantum-numbers mask of the enclosed interaction.

```

<Evaluators: public>+≡
  public :: evaluator_get_mask

<Evaluators: procedures>+≡
  function evaluator_get_mask (eval) result (mask)
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
    type(evaluator_t), intent(in), target :: eval
    allocate (mask (interaction_get_n_tot (eval%int)))
    mask = interaction_get_mask (eval%int)
  end function evaluator_get_mask

```

Extend the linking of interactions to evaluators.

```

(Evaluators: public)+≡
  public :: interaction_set_source_link
  public :: evaluator_set_source_link

(Evaluators: interfaces)+≡
  interface interaction_set_source_link
    module procedure interaction_set_source_link_eval
  end interface
  interface evaluator_set_source_link
    module procedure evaluator_set_source_link_int
    module procedure evaluator_set_source_link_eval
  end interface

(Evaluators: procedures)+≡
  subroutine interaction_set_source_link_eval (int, i, eval1, i1)
    type(interaction_t), intent(inout) :: int
    type(evaluator_t), intent(in), target :: eval1
    integer, intent(in) :: i, i1
    call interaction_set_source_link (int, i, eval1%int, i1)
  end subroutine interaction_set_source_link_eval

  subroutine evaluator_set_source_link_int (eval, i, int1, i1)
    type(evaluator_t), intent(inout) :: eval
    type(interaction_t), intent(in), target :: int1
    integer, intent(in) :: i, i1
    call interaction_set_source_link (eval%int, i, int1, i1)
  end subroutine evaluator_set_source_link_int

  subroutine evaluator_set_source_link_eval (eval, i, eval1, i1)
    type(evaluator_t), intent(inout) :: eval
    type(evaluator_t), intent(in), target :: eval1
    integer, intent(in) :: i, i1
    call interaction_set_source_link (eval%int, i, eval1%int, i1)
  end subroutine evaluator_set_source_link_eval

```

Receive from / send momenta to the linked interactions.

```

(Evaluators: public)+≡
  public :: evaluator_receive_momenta
  public :: evaluator_send_momenta

(Evaluators: procedures)+≡
  subroutine evaluator_receive_momenta (eval)
    type(evaluator_t), intent(inout) :: eval
    call interaction_receive_momenta (eval%int)
  end subroutine evaluator_receive_momenta

  subroutine evaluator_send_momenta (eval)
    type(evaluator_t), intent(in) :: eval
    call interaction_send_momenta (eval%int)
  end subroutine evaluator_send_momenta

```

Reassign external source links from one to another.

```

(Evaluators: public)+≡
  public :: evaluator_reassign_links

```

```

<Evaluators: interfaces>+≡
interface evaluator_reassign_links
  module procedure evaluator_reassign_links_eval
  module procedure evaluator_reassign_links_int
end interface

<Evaluators: procedures>+≡
subroutine evaluator_reassign_links_eval (eval, eval_src, eval_target)
  type(evaluator_t), intent(inout) :: eval
  type(evaluator_t), intent(in) :: eval_src
  type(evaluator_t), intent(in), target :: eval_target
  if (associated (eval%int_in1)) then
    if (interaction_get_tag (eval%int_in1) &
      == interaction_get_tag (eval_src%int)) then
      eval%int_in1 => eval_target%int
    end if
  end if
  if (associated (eval%int_in2)) then
    if (interaction_get_tag (eval%int_in2) &
      == interaction_get_tag (eval_src%int)) then
      eval%int_in2 => eval_target%int
    end if
  end if
  call interaction_reassign_links (eval%int, eval_src%int, eval_target%int)
end subroutine evaluator_reassign_links_eval

subroutine evaluator_reassign_links_int (eval, int_src, int_target)
  type(evaluator_t), intent(inout) :: eval
  type(interaction_t), intent(in) :: int_src
  type(interaction_t), intent(in), target :: int_target
  if (associated (eval%int_in1)) then
    if (interaction_get_tag (eval%int_in1) &
      == interaction_get_tag (int_src)) then
      eval%int_in1 => int_target
    end if
  end if
  if (associated (eval%int_in2)) then
    if (interaction_get_tag (eval%int_in2) &
      == interaction_get_tag (int_src)) then
      eval%int_in2 => int_target
    end if
  end if
  call interaction_reassign_links (eval%int, int_src, int_target)
end subroutine evaluator_reassign_links_int

```

Return flavor, momentum, and position of the first unstable particle present in the interaction.

```

<Evaluators: public>+≡
public :: evaluator_get_unstable_particle

<Evaluators: procedures>+≡
subroutine evaluator_get_unstable_particle (eval, flv, p, i)
  type(evaluator_t), intent(in) :: eval
  type(flavor_t), intent(out) :: flv

```

```

type(vector4_t), intent(out) :: p
integer, intent(out) :: i
call interaction_get_unstable_particle (eval%int, flv, p, i)
end subroutine evaluator_get_unstable_particle

```

### 7.7.8 Deleting the evaluator

Only the interaction component needs finalization.

```

<Evaluators: public>+≡
public :: evaluator_final

<Evaluators: procedures>+≡
elemental subroutine evaluator_final (eval)
type(evaluator_t), intent(inout) :: eval
call interaction_final (eval%int)
end subroutine evaluator_final

```

### 7.7.9 Creating an evaluator: identity

The identity evaluator creates a copy of the first input evaluator; the second input is not used.

All particles link back to the input evaluator and the internal relations are copied. As evaluation does take a shortcut by cloning the matrix elements, the pairing array is not used and does not have to be set up.

```

<Evaluators: public>+≡
public :: evaluator_init_identity

<Evaluators: interfaces>+≡
interface evaluator_init_identity
module procedure evaluator_init_identity_i
module procedure evaluator_init_identity_e
end interface

<Evaluators: procedures>+≡
subroutine evaluator_init_identity_i (eval, int)
type(evaluator_t), intent(out), target :: eval
type(interaction_t), intent(in), target :: int
integer :: n_in, n_out, n_vir, n_tot
integer :: i
integer, dimension(:), allocatable :: map
type(state_matrix_t), pointer :: state
type(state_iterator_t) :: it

eval%type = EVAL_IDENTITY
eval%int_in1 => int
nullify (eval%int_in2)
n_in = interaction_get_n_in (int)
n_out = interaction_get_n_out (int)
n_vir = interaction_get_n_vir (int)
n_tot = interaction_get_n_tot (int)

```

```

call interaction_init (eval%int, n_in, n_vir, n_out, &
  mask=interaction_get_mask (int), &
  resonant=interaction_get_resonance_flags (int))
do i = 1, n_tot
  call interaction_set_source_link (eval%int, i, int, i)
end do
allocate (map(n_tot))
map = /(i, i = 1, n_tot)/
call interaction_transfer_relations (int, eval%int, map)
state => interaction_get_state_matrix_ptr (int)
call state_iterator_init (it, state)
do while (state_iterator_is_valid (it))
  call interaction_add_state (eval%int, &
    state_iterator_get_quantum_numbers (it), &
    state_iterator_get_me_index (it))
  call state_iterator_advance (it)
end do
call interaction_freeze (eval%int)

end subroutine evaluator_init_identity_i

subroutine evaluator_init_identity_e (eval, eval1)
  type(evaluator_t), intent(out), target :: eval
  type(evaluator_t), intent(in), target :: eval1
  call evaluator_init_identity_i (eval, eval1%int)
end subroutine evaluator_init_identity_e

```

### 7.7.10 Creating an evaluator: quantum number sum

This evaluator operates on the diagonal of a density matrix and sums over the quantum numbers specified by the mask. The optional argument **drop** allows to drop a particle from the resulting density matrix. The handling of virtuals is not completely sane, especially in connection with dropping particles.

```

<Evaluators: public>+≡
  public :: evaluator_init_qn_sum

<Evaluators: interfaces>+≡
  interface evaluator_init_qn_sum
    module procedure evaluator_init_qn_sum_i
    module procedure evaluator_init_qn_sum_e
  end interface

<Evaluators: procedures>+≡
  subroutine evaluator_init_qn_sum_i (eval, int, qn_mask, drop)
    type(evaluator_t), intent(out), target :: eval
    type(interaction_t), target, intent(in) :: int
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    logical, intent(in), optional, dimension(:) :: drop
    type(state_iterator_t) :: it_old, it_new
    integer, dimension(:,:), allocatable :: pairing_proto
    integer, dimension(:), allocatable :: pairing_sizes
    integer, dimension(:), allocatable :: map
    integer :: n_in, n_out, n_vir, n_tot, n_me_old, n_me_new

```

```

integer :: i, j
type(state_matrix_t), pointer :: state_new, state_old
type(quantum_numbers_t), dimension(:), allocatable :: qn
logical :: matched
logical, dimension(size (qn_mask)) :: dropped
integer :: ndropped
integer, dimension(:), allocatable :: inotdropped
type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
logical, dimension(:), allocatable :: resonant

eval%type = EVAL_QN_SUM
eval%int_in1 => int
nullify (eval%int_in2)
if (present (drop)) then
    dropped = drop
else
    dropped = .false.
end if
ndropped = count (dropped)

n_in = interaction_get_n_in (int)
n_out = interaction_get_n_out (int) - ndropped
n_vir = interaction_get_n_vir (int)
n_tot = interaction_get_n_tot (int) - ndropped

allocate (inotdropped (n_tot))
i = 1
do j = 1, n_tot + ndropped
    if (dropped (j)) cycle
    inotdropped(i) = j
    i = i + 1
end do

allocate (mask(n_tot + ndropped))
mask = interaction_get_mask (int)
allocate (resonant(n_tot + ndropped))
resonant = interaction_get_resonance_flags (int)
call interaction_init (eval%int, n_in, n_vir, n_out, &
    mask = mask(inotdropped) .or. qn_mask(inotdropped), &
    resonant = resonant(inotdropped))
i = 1
do j = 1, n_tot + ndropped
    if (dropped(j)) cycle
    call interaction_set_source_link (eval%int, i, int, j)
    i = i + 1
end do
allocate (map(n_tot + ndropped))
i = 1
do j = 1, n_tot + ndropped
    if (dropped (j)) then
        map(j) = 0
    else
        map(j) = i
        i = i + 1
    end if
end do

```



```

        end if
    end do
    call interaction_transfer_relations (int, eval%int, map)

    n_me_old = interaction_get_n_matrix_elements (int)
    allocate (pairing_proto(n_me_old, n_me_old))
    allocate (pairing_sizes(n_me_old))
    pairing_sizes = 0
    state_old => interaction_get_state_matrix_ptr (int)
    state_new => interaction_get_state_matrix_ptr (eval%int)
    call state_iterator_init (it_old, state_old)
    allocate (qn(n_tot + ndropped))
    do while (state_iterator_is_valid (it_old))
        qn = state_iterator_get_quantum_numbers (it_old)
        if (.not. all (quantum_numbers_are_diagonal (qn))) then
            call state_iterator_advance (it_old)
            cycle
        end if
        matched = .false.
        call state_iterator_init (it_new, state_new)
        if (interaction_get_n_matrix_elements (eval%int) > 0) then
            do while (state_iterator_is_valid (it_new))
                if (all (qn(inotdropped) .match. &
                    state_iterator_get_quantum_numbers (it_new))) &
                then
                    matched = .true.
                    i = state_iterator_get_me_index (it_new)
                    exit
                end if
                call state_iterator_advance (it_new)
            end do
        end if
        if (.not. matched) then
            call interaction_add_state (eval%int, qn(inotdropped))
            i = interaction_get_n_matrix_elements (eval%int)
        end if
        pairing_sizes(i) = pairing_sizes(i) + 1
        pairing_proto(pairing_sizes(i), i) = &
            state_iterator_get_me_index (it_old)
!       print *, i, pairing_sizes(i), state_iterator_get_me_index (it_old)
        call state_iterator_advance (it_old)
    end do
    call interaction_freeze (eval%int)

    n_me_new = interaction_get_n_matrix_elements (eval%int)
    allocate (eval%pairing_array(n_me_new))
    do i = 1, n_me_new
!       print *, i, pairing_sizes(i)
        call pairing_array_init (eval%pairing_array(i), &
            pairing_sizes(i), .false., .false.)
        eval%pairing_array(i)%i1 = pairing_proto(:pairing_sizes(i), i)
    end do

end subroutine evaluator_init_qn_sum_i

```

```

subroutine evaluator_init_qn_sum_e (eval, eval1, qn_mask, drop)
  type(evaluator_t), intent(out) :: eval
  type(evaluator_t), intent(in), target :: eval1
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  logical, dimension(:), optional, intent(in) :: drop
  call evaluator_init_qn_sum_i (eval, eval1%int, qn_mask, drop)
end subroutine

```

### 7.7.11 Evaluation

When the input interactions (which are pointed to in the pairings stored within the evaluator) are filled with values, we can activate the evaluator, i.e., calculate the result values which are stored in the interaction.

The evaluation of matrix elements can be done in parallel. A `forall` construct is not appropriate, however. We would need `do concurrent` here. Nevertheless, the evaluation functions are marked as `pure`.

```

<Evaluators: public>+≡
  public :: evaluator_evaluate

<Evaluators: procedures>+≡
  subroutine evaluator_evaluate (eval)
    type(evaluator_t), intent(inout), target :: eval
    integer :: i
    select case (eval%type)
    case (EVAL_PRODUCT)
      do i = 1, size(eval%pairing_array)
        call interaction_evaluate_product (eval%int, i, &
          eval%int_in1, eval%int_in2, &
          eval%pairing_array(i)%i1, eval%pairing_array(i)%i2)
      end do
    case (EVAL_SQUARE_WITH_COLOR_FACTORS)
      do i = 1, size(eval%pairing_array)
        call interaction_evaluate_product_cf (eval%int, i, &
          eval%int_in1, eval%int_in1, &
          eval%pairing_array(i)%i1, eval%pairing_array(i)%i2, &
          eval%pairing_array(i)%factor)
      end do
    case (EVAL_SQUARED_FLOWS)
      do i = 1, size(eval%pairing_array)
        call interaction_evaluate_square_c (eval%int, i, &
          eval%int_in1, &
          eval%pairing_array(i)%i1)
      end do
    case (EVAL_COLOR_CONTRACTION)
      do i = 1, size(eval%pairing_array)
        call interaction_evaluate_sum (eval%int, i, &
          eval%int_in1, &
          eval%pairing_array(i)%i1)
      end do
    case (EVAL_IDENTITY)
      call interaction_set_matrix_element (eval%int, eval%int_in1)
    end select
  end subroutine

```

```

case (EVAL_QN_SUM)
  do i = 1, size (eval%pairing_array)
    call interaction_evaluate_sum (eval%int, i, &
      eval%int_in1, eval%pairing_array(i)%i1)
  end do
end select
end subroutine evaluator_evaluate

```

### 7.7.12 Test

Test: Create two interactions. The interactions are twofold connected. The first connection has a helicity index that is kept, the second connection has a helicity index that is summed over. Concatenate the interactions in an evaluator, which thus contains a result interaction. Fill the input interactions with values, activate the evaluator and print the result.

*(Evaluators: public)+≡*

```
public :: evaluator_test
```

*(Evaluators: procedures)+≡*

```

subroutine evaluator_test (mdl)
  type(model_t), intent(in), target :: mdl
  call evaluator_test1 (mdl)
  call evaluator_test2 (mdl)
  call evaluator_test3 (mdl)
end subroutine evaluator_test

subroutine evaluator_test1 (mdl)
  type(model_t), intent(in), target :: mdl
  type(interaction_t), target :: int_qqtt, int_tbw, int1, int2
  type(flavor_t), dimension(:), allocatable :: flv
  type(color_t), dimension(:), allocatable :: col
  type(helicity_t), dimension(:), allocatable :: hel
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  integer :: f, c, h1, h2, h3
  type(vector4_t), dimension(4) :: p
  type(vector4_t), dimension(2) :: q
  type(quantum_numbers_mask_t) :: qn_mask_conn
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask2
  type(evaluator_t), target :: eval, eval2, eval3
  print *, "*** Evaluator for matrix product"
  print *, "*** Construct interaction for qq -> tt"
  call interaction_init (int_qqtt, 2, 0, 2, set_relations=.true.)
  allocate (flv (4), col (4), hel (4), qn (4))
  allocate (qn_mask2 (4))
  do c = 1, 2
    select case (c)
    case (1)
      call color_init_col_acl (col, (/ 1, 0, 1, 0 /), (/ 0, 2, 0, 2 /))
    case (2)
      call color_init_col_acl (col, (/ 1, 0, 2, 0 /), (/ 0, 1, 0, 2 /))
    end select
  do f = 1, 2

```

```

call flavor_init (flv, (/f, -f, 6, -6/), mdl)
do h1 = -1, 1, 2
  call helicity_init (hel(3), h1)
  do h2 = -1, 1, 2
    call helicity_init (hel(4), h2)
    call quantum_numbers_init (qn, flv, col, hel)
    call interaction_add_state (int_qqtt, qn)
  end do
end do
end do
call interaction_freeze (int_qqtt)
deallocate (flv, col, hel, qn)
print *, "*** Construct interaction for t -> bW"
call interaction_init (int_tbw, 1, 0, 2, set_relations=.true.)
allocate (flv (3), col (3), hel (3), qn (3))
call flavor_init (flv, (/ 6, 5, 24 /), mdl)
call color_init_col_acl (col, (/ 1, 1, 0 /), (/ 0, 0, 0 /))
do h1 = -1, 1, 2
  call helicity_init (hel(1), h1)
  do h2 = -1, 1, 2
    call helicity_init (hel(2), h2)
    do h3 = -1, 1
      call helicity_init (hel(3), h3)
      call quantum_numbers_init (qn, flv, col, hel)
      call interaction_add_state (int_tbw, qn)
    end do
  end do
end do
call interaction_freeze (int_tbw)
deallocate (flv, col, hel, qn)
print *, "*** Link interactions"
call interaction_set_source_link (int_tbw, 1, int_qqtt, 3)
qn_mask_conn = new_quantum_numbers_mask (.false., .false., .true.)
print *, "*** Show input"
call interaction_write (int_qqtt)
print *
call interaction_write (int_tbw)
print *
print *, "*** Evaluate product"
call evaluator_init_product &
  (eval, int_qqtt, int_tbw, qn_mask_conn)
call evaluator_write (eval)

call interaction_init (int1, 2, 0, 2, set_relations=.true.)
call interaction_init (int2, 1, 0, 2, set_relations=.true.)
p(1) = vector4_moving (1000._default, 1000._default, 3)
p(2) = vector4_moving (200._default, 200._default, 2)
p(3) = vector4_moving (100._default, 200._default, 1)
p(4) = p(1) - p(2) - p(3)
call interaction_set_momenta (int1, p)
q(1) = vector4_moving (50._default, -50._default, 3)
q(2) = p(2) + p(4) - q(1)
call interaction_set_momenta (int2, q, outgoing=.true.)

```

```

call interaction_set_matrix_element &
    (int1, ((2._default,0._default), (4._default,1._default), (-3._default,0._default)))
call interaction_set_matrix_element &
    (int2, ((-3._default,0._default), (0._default,1._default), (1._default,2._default)))
call evaluator_receive_momenta (eval)
call evaluator_evaluate (eval)
call interaction_write (int1)
print *
call interaction_write (int2)
print *
call evaluator_write (eval)
print *
call interaction_final (int1)
call interaction_final (int2)
call evaluator_final (eval)

print *
print *, "*** Evaluator for matrix square"
allocate (flv(4), col(4), qn(4))
call interaction_init (int1, 2, 0, 2, set_relations=.true.)
call flavor_init (flv, (/1, -1, 21, 21/), mdl)
call color_init (col(1), (/1/))
call color_init (col(2), (/ -2/))
call color_init (col(3), (/2, -3/))
call color_init (col(4), (/3, -1/))
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call color_init (col(3), (/3, -1/))
call color_init (col(4), (/2, -3/))
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call color_init (col(3), (/2, -1/))
call color_init (col(4), .true.)
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call interaction_freeze (int1)
! qn_mask2 = all false (default)
call evaluator_init_square (eval, int1, qn_mask2, nc=3)
call evaluator_init_square_nondiag (eval2, int1, qn_mask2)
qn_mask2 = new_quantum_numbers_mask (.false., .true., .true.)
call evaluator_init_square_diag (eval3, eval%int, qn_mask2)
call interaction_set_matrix_element &
    (int1, ((2._default,0._default), (4._default,1._default), (-3._default,0._default)))
call interaction_set_momenta (int1, p)
call interaction_write (int1)
print *
call evaluator_receive_momenta (eval)
call evaluator_evaluate (eval)
call evaluator_write (eval)
print *
call evaluator_receive_momenta (eval2)
call evaluator_evaluate (eval2)
call evaluator_write (eval2)
print *

```

```

    call evaluator_receive_momenta (eval3)
    call evaluator_evaluate (eval3)
    call evaluator_write (eval3)
    call interaction_final (int1)
    call evaluator_final (eval)
    call evaluator_final (eval2)
    call evaluator_final (eval3)
end subroutine evaluator_test1

subroutine evaluator_test2 (mdl)
  type(model_t), target, intent(in) :: mdl
  type(interaction_t), target :: int
  integer :: h1, h2, h3, h4
  type(helicity_t), dimension(4) :: hel
  type(color_t), dimension(4) :: col
  type(flavor_t), dimension(4) :: flv
  type(quantum_numbers_t), dimension(4) :: qn
  type(vector4_t), dimension(4) :: p
  type(evaluator_t) :: eval
  integer :: i
  print *, "*** Creating interaction for e+ e- -> W+ W-"
  call flavor_init (flv, (/11, -11, 24, -24/), mdl)
  do i = 1, 4
    call color_init (col (i))
  end do
  call interaction_init (int, 2, 0, 2, set_relations=.true.)
  do h1 = -1, 1, 2
    call helicity_init (hel(1), h1)
  do h2 = -1, 1, 2
    call helicity_init (hel(2), h2)
  do h3 = -1, 1
    call helicity_init (hel(3), h3)
  do h4 = -1, 1
    call helicity_init (hel(4), h4)
    call quantum_numbers_init (qn, flv, col, hel)
    call interaction_add_state (int, qn)
  end do
  end do
  end do
  call interaction_freeze (int)
  call interaction_set_matrix_element (int, &
    (/ (cmplx (i, kind=default), i = 1, 36) /))
  p(1) = vector4_moving (1000._default, 1000._default, 3)
  p(2) = vector4_moving (1000._default, -1000._default, 3)
  p(3) = vector4_moving (1000._default, &
    sqrt (1E6_default - 80._default**2), 3)
  p(4) = p(1) + p(2) - p(3)
  call interaction_set_momenta (int, p)
  print *, "*** Setting up evaluator"
  call evaluator_init_identity (eval, int)
  print *, "*** Transferring momenta and evaluating"
  call evaluator_receive_momenta (eval)
  call evaluator_evaluate (eval)

```

```

print *, "*****"
print *, "    Interaction dump"
print *, "*****"
call interaction_write (int)
print *
print *, "*****"
print *, "    Evaluator dump"
print *, "*****"
call evaluator_write (eval)
print *
print *, "*** cleaning up"
call interaction_final (int)
call evaluator_final (eval)
end subroutine evaluator_test2

subroutine evaluator_test3 (mdl)
  type(model_t), target, intent(in) :: mdl
  type(interaction_t), target :: int
  integer :: h1, h2, h3, h4
  type(helicity_t), dimension(4) :: hel
  type(color_t), dimension(4) :: col
  type(flavor_t), dimension(4) :: flv1, flv2
  type(quantum_numbers_t), dimension(4) :: qn
  type(vector4_t), dimension(4) :: p
  type(evaluator_t) :: eval1, eval2, eval3
  type(quantum_numbers_mask_t), dimension(4) :: qn_mask
  integer :: i
  print *, "*** Creating interaction for e+/mu+ e-/mu- -> W+ W-"
  call flavor_init (flv1, (/11, -11, 24, -24/), mdl)
  call flavor_init (flv2, (/13, -13, 24, -24/), mdl)
  do i = 1, 4
    call color_init (col (i))
  end do
  call interaction_init (int, 2, 0, 2, set_relations=.true.)
  do h1 = -1, 1, 2
    call helicity_init (hel(1), h1)
    do h2 = -1, 1, 2
      call helicity_init (hel(2), h2)
      do h3 = -1, 1
        call helicity_init (hel(3), h3)
        do h4 = -1, 1
          call helicity_init (hel(4), h4)
          call quantum_numbers_init (qn, flv1, col, hel)
          call interaction_add_state (int, qn)
          call quantum_numbers_init (qn, flv2, col, hel)
          call interaction_add_state (int, qn)
        end do
      end do
    end do
  end do
  call interaction_freeze (int)
  call interaction_set_matrix_element (int, &
    ((cplx (1, kind=default), i = 1, 72)/))
  p(1) = vector4_moving (1000._default, 1000._default, 3)

```

```

p(2) = vector4_moving (1000._default, -1000._default, 3)
p(3) = vector4_moving (1000._default, &
    sqrt (1E6_default - 80._default**2), 3)
p(4) = p(1) + p(2) - p(3)
call interaction_set_momenta (int, p)
print *, "*** Setting up evaluators"
call quantum_numbers_mask_init (qn_mask, .false., .true., .true.)
call evaluator_init_qn_sum (eval1, int, qn_mask)
call quantum_numbers_mask_init (qn_mask, .true., .true., .true.)
call evaluator_init_qn_sum (eval2, int, qn_mask)
call quantum_numbers_mask_init (qn_mask, .false., .true., .false.)
call evaluator_init_qn_sum (eval3, int, qn_mask, &
    (/ .false., .false., .false., .true. /))
print *, "*** Transferring momenta and evaluating"
call evaluator_receive_momenta (eval1)
call evaluator_evaluate (eval1)
call evaluator_receive_momenta (eval2)
call evaluator_evaluate (eval2)
call evaluator_receive_momenta (eval3)
call evaluator_evaluate (eval3)
print *, "*****"
print *, "    Interaction dump"
print *, "*****"
call interaction_write (int)
print *
print *, "*****"
print *, "    Evaluator dump --- spin sum"
print *, "*****"
call evaluator_write (eval1)
call interaction_write (evaluator_get_int_ptr (eval1))
print *, "*****"
print *, "    Evaluator dump --- spin / flavor sum"
print *, "*****"
call evaluator_write (eval2)
call interaction_write (evaluator_get_int_ptr (eval2))
print *, "*****"
print *, "    Evaluator dump --- flavor sum, drop last W"
print *, "*****"
call evaluator_write (eval3)
call interaction_write (evaluator_get_int_ptr (eval3))
print *
print *, "*** cleaning up"
call interaction_final (int)
call evaluator_final (eval1)
call evaluator_final (eval2)
call evaluator_final (eval3)
end subroutine evaluator_test3

```



## Chapter 8

# Particles

In this chapter, we deal with particles which have well-defined quantum numbers. While within interactions, all correlations are manifest, a particle array is derived by selecting a particular quantum number set. This involves tracing over all other particles, as far as polarization is concerned. Thus, a particle has definite flavor, color, and a single-particle density matrix for polarization.

### 8.1 Polarization

Particle polarization is determined by a particular quantum state which has just helicity information. For defining polarizations, we adopt the phase convention for a spin-1/2 particle that

$$\rho = \frac{1}{2}(1 + \vec{\alpha} \cdot \vec{\sigma}) \quad (8.1)$$

with the polarization axis  $\vec{\alpha}$ . Using this, we define

1. Trivial polarization:  $\vec{\alpha} = 0$ . [This is unpolarized, but distinct from the particular undefined polarization matrix which has the same meaning.]
2. Circular polarization:  $\vec{\alpha}$  points in  $\pm z$  direction.
3. Transversal polarization:  $\vec{\alpha}$  points orthogonal to the  $z$  direction, with a phase  $\phi$  that is 0 for the  $x$  axis, and  $\pi/2 = 90^\circ$  for the  $y$  axis. For antiparticles, the phase switches sign, corresponding to complex conjugation.
4. Axis polarization, where we explicitly give  $\vec{\alpha}$ .

For higher spin, we retain this definition, but apply it to the two components with maximum and minimum weight. For massless particles, this is sufficient. For massive particles, we add the possibilities:

5. Longitudinal polarization: Only the 0-component is set. This is possible only for bosons.
6. Diagonal polarization: Explicitly specify all components in the helicity basis.

Obviously, this does not exhaust the possible density matrices for higher spin, but it should cover all practical applications.

```

<polarizations.f90>≡
  <File header>

  module polarizations

    <Use kinds>
    <Use strings>
    use constants, only: imago !NODEP!
    <Use file utils>
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices

    <Standard module head>

    <Polarizations: public>

    <Polarizations: types>

    <Polarizations: interfaces>

    contains

    <Polarizations: procedures>

  end module polarizations

```

### 8.1.1 The polarization type

This is not an extension, but rather a restriction of the quantum state. Flavor and color are ignored, there is just a one-particle helicity density matrix.

```

<Polarizations: public>≡
  public :: polarization_t

<Polarizations: types>≡
  type :: polarization_t
    logical :: polarized = .false.
    integer :: spin_type = 0
    integer :: multiplicity = 0
    type(state_matrix_t) :: state
  end type polarization_t

```

### 8.1.2 Basic initializer and finalizer

We need the particle flavor for determining the allowed helicity values. The density matrix is not set, but prepared to be filled later. This is private.

```

<Polarizations: procedures>≡
  elemental subroutine polarization_init (pol, flv)
    type(polarization_t), intent(out) :: pol
    type(flavor_t), intent(in) :: flv
    pol%spin_type = flavor_get_spin_type (flv)
    pol%multiplicity = flavor_get_multiplicity (flv)
    call state_matrix_init (pol%state, store_values = .true.)
  end subroutine polarization_init

```

The finalizer has to be public. The quantum state contains memory allocated to pointers.

```

<Polarizations: public>+≡
  public :: polarization_final

<Polarizations: procedures>+≡
  elemental subroutine polarization_final (pol)
    type(polarization_t), intent(inout) :: pol
    call state_matrix_final (pol%state)
  end subroutine polarization_final

```

### 8.1.3 I/O

```

<Polarizations: public>+≡
  public :: polarization_write

<Polarizations: procedures>+≡
  subroutine polarization_write (pol, unit)
    type(polarization_t), intent(in) :: pol
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A,I1,A,I1,A)") &
      "Polarization: [spin_type = ", pol%spin_type, &
      ", mult = ", pol%multiplicity, "]"
    call state_matrix_write (pol%state, unit=unit)
  end subroutine polarization_write

```

Defined assignment: deep copy

```

<Polarizations: public>+≡
  public :: assignment(=)

<Polarizations: interfaces>≡
  interface assignment(=)
    module procedure polarization_assign
  end interface

```

```

<Polarizations: procedures>+≡
  subroutine polarization_assign (pol_out, pol_in)
    type(polarization_t), intent(out) :: pol_out
    type(polarization_t), intent(in) :: pol_in
    pol_out%polarized = pol_in%polarized
    pol_out%spin_type = pol_in%spin_type

```

```

    pol_out%multiplicity = pol_in%multiplicity
    pol_out%state = pol_in%state
end subroutine polarization_assign

```

Binary I/O.

```

<Polarizations: public>+≡
    public :: polarization_write_raw
    public :: polarization_read_raw

<Polarizations: procedures>+≡
    subroutine polarization_write_raw (pol, u)
        type(polarization_t), intent(in) :: pol
        integer, intent(in) :: u
        write (u) pol%polarized
        write (u) pol%spin_type
        write (u) pol%multiplicity
        call state_matrix_write_raw (pol%state, u)
    end subroutine polarization_write_raw

    subroutine polarization_read_raw (pol, u, iostat)
        type(polarization_t), intent(out) :: pol
        integer, intent(in) :: u
        integer, intent(out), optional :: iostat
        read (u, iostat=iostat) pol%polarized
        read (u, iostat=iostat) pol%spin_type
        read (u, iostat=iostat) pol%multiplicity
        call state_matrix_read_raw (pol%state, u, iostat=iostat)
    end subroutine polarization_read_raw

```

#### 8.1.4 Accessing contents

Return true if the particle is polarized. This is the case if the first (and only) entry in the quantum state has undefined helicity.

```

<Polarizations: public>+≡
    public :: polarization_is_polarized

<Polarizations: procedures>+≡
    elemental function polarization_is_polarized (pol) result (polarized)
        logical :: polarized
        type(polarization_t), intent(in) :: pol
        polarized = pol%polarized
    end function polarization_is_polarized

```

Return true if the polarization is diagonal, i.e., all entries in the density matrix are diagonal.

```

<Polarizations: public>+≡
    public :: polarization_is_diagonal

<Polarizations: interfaces>+≡
    interface polarization_is_diagonal
        module procedure polarization_is_diagonal0
        module procedure polarization_is_diagonal1
    end interface

```

```
end interface
```

*(Polarizations: procedures)*+≡

```
function polarization_is_diagonal0 (pol) result (diagonal)
  logical :: diagonal
  type(polarization_t), intent(in) :: pol
  type(state_iterator_t) :: it
  diagonal = .true.
  call state_iterator_init (it, pol%state)
  do while (state_iterator_is_valid (it))
    diagonal = all (quantum_numbers_are_diagonal &
      (state_iterator_get_quantum_numbers (it)))
    if (.not. diagonal) exit
    call state_iterator_advance (it)
  end do
end function polarization_is_diagonal0

function polarization_is_diagonal1 (pol) result (diagonal)
  type(polarization_t), dimension(:), intent(in) :: pol
  logical, dimension(size(pol)) :: diagonal
  integer :: i
  do i = 1, size (pol)
    diagonal(i) = polarization_is_diagonal0 (pol(i))
  end do
end function polarization_is_diagonal1
```

### 8.1.5 Initialization from state matrix

Here, the state matrix is already known (but not necessarily normalized). The result will be either unpolarized, or a normalized spin density matrix.

*(Polarizations: public)*+≡

```
public :: polarization_init_state_matrix
```

*(Polarizations: procedures)*+≡

```
subroutine polarization_init_state_matrix (pol, state)
  type(polarization_t), intent(out) :: pol
  type(state_matrix_t), intent(in), target :: state
  type(state_iterator_t) :: it
  type(flavor_t) :: flv
  type(helicity_t) :: hel
  type(quantum_numbers_t), dimension(1) :: qn
  complex(default) :: value, t
  call state_iterator_init (it, state)
  flv = state_iterator_get_flavor (it, 1)
  hel = state_iterator_get_helicity (it, 1)
  if (helicity_is_defined (hel)) then
    call polarization_init (pol, flv)
    pol%polarized = .true.
    t = 0
  do while (state_iterator_is_valid (it))
    hel = state_iterator_get_helicity (it, 1)
    call quantum_numbers_init (qn(1), hel)
```

```

        value = state_iterator_get_matrix_element (it)
        call state_matrix_add_state (pol%state, qn, value=value)
        if (helicity_is_diagonal (hel)) t = t + value
        call state_iterator_advance (it)
    end do
    call state_matrix_freeze (pol%state)
    if (t /= 0) call state_matrix_renormalize (pol%state, 1._default / t)
else
    call polarization_init_unpolarized (pol, flv)
end if
end subroutine polarization_init_state_matrix

```

### 8.1.6 Specific initializers

Unpolarized particle, no helicity labels in the density matrix. The value is specified as  $1/N$ , where  $N$  is the multiplicity.

Exception: for left-handed or right-handed particles (neutrinos), polarization is always circular with fraction unity.

```

(Polarizations: public)+≡
    public :: polarization_init_unpolarized

(Polarizations: procedures)+≡
    subroutine polarization_init_unpolarized (pol, flv)
        type(polarization_t), intent(inout) :: pol
        type(flavor_t), intent(in) :: flv
        type(quantum_numbers_t), dimension(1) :: qn
        complex(default) :: value
        if (flavor_is_left_handed (flv)) then
            call polarization_init_circular (pol, flv, -1._default)
        else if (flavor_is_right_handed (flv)) then
            call polarization_init_circular (pol, flv, 1._default)
        else
            call polarization_init (pol, flv)
            value = 1._default / flavor_get_multiplicity (flv)
            call state_matrix_add_state (pol%state, qn)
            call state_matrix_freeze (pol%state)
            call state_matrix_set_matrix_element (pol%state, value)
        end if
    end subroutine polarization_init_unpolarized

```

Unpolarized particle, but explicit density matrix with helicity states allocated according to given flavor. Note that fermions have even spin type, bosons odd. The spin density matrix entries are scaled by **fraction**. This is used for initializing other polarizations:

$$\rho(f) = \frac{|f|}{N} \mathbf{1}.$$

```

(Polarizations: public)+≡
    public :: polarization_init_trivial

(Polarizations: procedures)+≡
    subroutine polarization_init_trivial (pol, flv, fraction)
        type(polarization_t), intent(out) :: pol

```

```

type(flavor_t), intent(in) :: flv
real(default), intent(in), optional :: fraction
type(helicity_t) :: hel
type(quantum_numbers_t), dimension(1) :: qn
integer :: h, hmax
logical :: fermion
complex(default) :: value
call polarization_init (pol, flv)
pol%polarized = .true.
if (present (fraction)) then
    value = fraction / pol%multiplicity
else
    value = 1._default / pol%multiplicity
end if
fermion = mod (pol%spin_type, 2) == 0
hmax = pol%spin_type / 2
select case (pol%multiplicity)
case (1)
    if (flavor_is_left_handed (flv)) then
        call helicity_init (hel, -hmax)
    else if (flavor_is_right_handed (flv)) then
        call helicity_init (hel, hmax)
    else
        call helicity_init (hel, 0)
    end if
    call quantum_numbers_init (qn(1), hel)
    call state_matrix_add_state (pol%state, qn)
case (2)
    do h = -hmax, hmax, 2*hmax
        call helicity_init (hel, h)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
    end do
case default
    do h = -hmax, hmax
        if (fermion .and. h == 0) cycle
        call helicity_init (hel, h)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
    end do
end select
call state_matrix_freeze (pol%state)
call state_matrix_set_matrix_element (pol%state, value)
end subroutine polarization_init_trivial

```

The following three modes are useful mainly for spin-1/2 particle and massless particles of any nonzero spin. Only the highest-weight components are filled.

Circular polarization: The density matrix of the two highest-weight states

is

$$\rho(f) = \frac{1 - |f|}{2} \mathbf{1} + |f| \times \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, & f > 0; \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, & f < 0, \end{cases}$$

If the polarization fraction  $|f|$  is unity, we need only one entry in the density matrix.

```

(Polarizations: public)+≡
    public :: polarization_init_circular

(Polarizations: procedures)+≡
    subroutine polarization_init_circular (pol, flv, fraction)
        type(polarization_t), intent(out) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), intent(in) :: fraction
        type(helicity_t), dimension(2) :: hel
        type(quantum_numbers_t), dimension(1) :: qn
        complex(default) :: value
        integer :: hmax
        call polarization_init (pol, flv)
        pol%polarized = .true.
        hmax = pol%spin_type / 2
        call helicity_init (hel(1), hmax)
        call helicity_init (hel(2), -hmax)
        if (abs (fraction) /= 1) then
            value = (1 + fraction) / 2
            call quantum_numbers_init (qn(1), hel(1))
            call state_matrix_add_state (pol%state, qn, value=value)
            value = (1 - fraction) / 2
            call quantum_numbers_init (qn(1), hel(2))
            call state_matrix_add_state (pol%state, qn, value=value)
        else
            value = abs (fraction)
            if (fraction > 0) then
                call quantum_numbers_init (qn(1), hel(1))
            else
                call quantum_numbers_init (qn(1), hel(2))
            end if
            call state_matrix_add_state (pol%state, qn, value=value)
        end if
        call state_matrix_freeze (pol%state)
    end subroutine polarization_init_circular

```

Transversal polarization is analogous to circular, but we get a density matrix

$$\rho(f, \phi) = \frac{1 - |f|}{2} \mathbf{1} + \frac{|f|}{2} \begin{pmatrix} 1 & e^{-i\phi} \\ e^{i\phi} & 1 \end{pmatrix}.$$

The phase is  $\phi = 0$  for the  $x$ -axis,  $\phi = 90^\circ$  for the  $y$  axis as polarization vector. For an antiparticle, the phase switches sign, and for  $f < 0$ , the off-diagonal elements switch sign.

```

(Polarizations: public)+≡
    public :: polarization_init_transversal

```



```

<Polarizations: procedures>+≡
subroutine polarization_init_transversal (pol, flv, phi, fraction)
  type(polarization_t), intent(inout) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: phi, fraction
  call polarization_init_axis &
    (pol, flv, fraction * (/ cos (phi), sin (phi), 0._default/))
end subroutine polarization_init_transversal

```

For axis polarization, we again set only the entries with maximum weight.

$$\rho(f, \phi) = \frac{1}{2} \begin{pmatrix} 1 + \alpha_3 & \alpha_1 - i\alpha_2 \\ \alpha_1 + i\alpha_2 & 1 - \alpha_3 \end{pmatrix}.$$

For an antiparticle,  $\alpha_2$  switches sign (complex conjugate).

```

<Polarizations: public>+≡
public :: polarization_init_axis

<Polarizations: procedures>+≡
subroutine polarization_init_axis (pol, flv, alpha)
  type(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), dimension(3), intent(in) :: alpha
  type(quantum_numbers_t), dimension(1) :: qn
  type(helicity_t), dimension(2,2) :: hel
  complex(default), dimension(2,2) :: value
  integer :: hmax
  call polarization_init (pol, flv)
  pol%polarized = .true.
  hmax = pol%spin_type / 2
  call helicity_init (hel(1,1), hmax, hmax)
  call helicity_init (hel(1,2), hmax, -hmax)
  call helicity_init (hel(2,1), -hmax, hmax)
  call helicity_init (hel(2,2), -hmax, -hmax)
  value(1,1) = (1 + alpha(3)) / 2
  value(2,2) = (1 - alpha(3)) / 2
  if (flavor_is_antiparticle (flv)) then
    value(1,2) = (alpha(1) + imago * alpha(2)) / 2
  else
    value(1,2) = (alpha(1) - imago * alpha(2)) / 2
  end if
  value(2,1) = conjg (value(1,2))
  if (value(1,1) /= 0) then
    call quantum_numbers_init (qn(1), hel(1,1))
    call state_matrix_add_state (pol%state, qn, value=value(1,1))
  end if
  if (value(2,2) /= 0) then
    call quantum_numbers_init (qn(1), hel(2,2))
    call state_matrix_add_state (pol%state, qn, value=value(2,2))
  end if
  if (value(1,2) /= 0) then
    call quantum_numbers_init (qn(1), hel(1,2))
    call state_matrix_add_state (pol%state, qn, value=value(1,2))
    call quantum_numbers_init (qn(1), hel(2,1))

```

```

        call state_matrix_add_state (pol%state, qn, value=value(2,1))
    end if
    call state_matrix_freeze (pol%state)
end subroutine polarization_init_axis

```

This version specifies the polarization axis in terms of  $r$  (polarization degree) and  $\theta, \phi$  (polar and azimuthal angles).

If one of the angles is a nonzero multiple of  $\pi$ , roundoff errors typically will result in tiny contributions to unwanted components. Therefore, include a catch for small numbers.

```

<Polarizations: public>+≡
    public :: polarization_init_angles

<Polarizations: procedures>+≡
    subroutine polarization_init_angles (pol, flv, r, theta, phi)
        type(polarization_t), intent(out) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), intent(in) :: r, theta, phi
        real(default), dimension(3) :: alpha
        real(default), parameter :: eps = 10 * epsilon (1._default)
        alpha(1) = r * sin (theta) * cos (phi)
        alpha(2) = r * sin (theta) * sin (phi)
        alpha(3) = r * cos (theta)
        where (abs (alpha) < eps) alpha = 0
        call polarization_init_axis (pol, flv, alpha)
    end subroutine polarization_init_angles

```

Longitudinal polarization is defined only for massive bosons. Only the zero component is filled. Otherwise, unpolarized.

```

<Polarizations: public>+≡
    public :: polarization_init_longitudinal

<Polarizations: procedures>+≡
    subroutine polarization_init_longitudinal (pol, flv, fraction)
        type(polarization_t), intent(out) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), intent(in) :: fraction
        integer :: spin_type, multiplicity
        type(helicity_t) :: hel
        type(quantum_numbers_t), dimension(1) :: qn
        complex(default) :: value
        integer :: n_values
        value = abs (fraction)
        spin_type = flavor_get_spin_type (flv)
        multiplicity = flavor_get_multiplicity (flv)
        if (mod (spin_type, 2) == 1 .and. multiplicity > 2) then
            if (fraction /= 1) then
                call polarization_init_trivial (pol, flv, 1 - fraction)
                n_values = state_matrix_get_n_matrix_elements (pol%state)
                call state_matrix_add_to_matrix_element &
                    (pol%state, n_values/2 + 1, value)
            else
                call polarization_init (pol, flv)
            end if
        end if
    end subroutine polarization_init_longitudinal

```

```

        pol%polarized = .true.
        call helicity_init (hel, 0)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
        call state_matrix_freeze (pol%state)
        call state_matrix_set_matrix_element (pol%state, value)
    end if
else
    call polarization_init_unpolarized (pol, flv)
end if
end subroutine polarization_init_longitudinal

```

This is diagonal polarization: we specify all components explicitly. We use only the positive components. The sum is normalized to unity. We assume that the length of `alpha` is equal to the particle multiplicity.

```

<Polarizations: public>+≡
    public :: polarization_init_diagonal

<Polarizations: procedures>+≡
    subroutine polarization_init_diagonal (pol, flv, alpha)
        type(polarization_t), intent(inout) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), dimension(:), intent(in) :: alpha
        type(helicity_t) :: hel
        type(quantum_numbers_t), dimension(1) :: qn
        logical, dimension(size(alpha)) :: mask
        real(default) :: norm
        complex(default), dimension(:), allocatable :: value
        logical :: fermion
        integer :: h, hmax, i
        mask = alpha > 0
        norm = sum (alpha, mask); if (norm == 0) norm = 1
        allocate (value (count (mask)))
        value = pack (alpha / norm, mask)
        call polarization_init (pol, flv)
        pol%polarized = .true.
        fermion = mod (pol%spin_type, 2) == 0
        hmax = pol%spin_type / 2
        i = 0
        select case (pol%multiplicity)
        case (1)
            if (flavor_is_left_handed (flv)) then
                call helicity_init (hel, -hmax)
            else if (flavor_is_right_handed (flv)) then
                call helicity_init (hel, hmax)
            else
                call helicity_init (hel, 0)
            end if
            call quantum_numbers_init (qn(1), hel)
            call state_matrix_add_state (pol%state, qn)
        case (2)
            do h = -hmax, hmax, 2*hmax
                i = i + 1
                if (mask(i)) then

```

```

        call helicity_init (hel, h)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
    end if
end do
case default
do h = -hmax, hmax
    if (fermion .and. h == 0) cycle
    i = i + 1
    if (mask(i)) then
        call helicity_init (hel, h)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
    end if
end do
end select
call state_matrix_freeze (pol%state)
call state_matrix_set_matrix_element (pol%state, value)
end subroutine polarization_init_diagonal

```

Generic polarization: we generate all possible density matrix entries, but the values are left zero.

*<Polarizations: public>+≡*

```
public :: polarization_init_generic
```

*<Polarizations: procedures>+≡*

```

subroutine polarization_init_generic (pol, flv)
    type(polarization_t), intent(out) :: pol
    type(flavor_t), intent(in) :: flv
    type(helicity_t) :: hel
    type(quantum_numbers_t), dimension(1) :: qn
    logical :: fermion
    integer :: hmax, h1, h2
    call polarization_init (pol, flv)
    pol%polarized = .true.
    fermion = mod (pol%spin_type, 2) == 0
    hmax = pol%spin_type / 2
    select case (pol%multiplicity)
    case (1)
        if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
        else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
        else
            call helicity_init (hel, 0)
        end if
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn)
    case (2)
        do h1 = -hmax, hmax, 2*hmax
            do h2 = -hmax, hmax, 2*hmax
                call helicity_init (hel, h1, h2)
                call quantum_numbers_init (qn(1), hel)
                call state_matrix_add_state (pol%state, qn)
            end do
        end do
    end select
end subroutine polarization_init_generic

```

```

        end do
    end do
case default
    do h1 = -hmax, hmax
        if (fermion .and. h1 == 0) cycle
        do h2 = -hmax, hmax
            if (fermion .and. h2 == 0) cycle
            call helicity_init (hel, h1, h2)
            call quantum_numbers_init (qn(1), hel)
            call state_matrix_add_state (pol%state, qn)
        end do
    end do
end select
call state_matrix_freeze (pol%state)
end subroutine polarization_init_generic

```

### 8.1.7 Operations

Combine polarization states by computing the outer product of the state matrices.

```

(Polarizations: public)+≡
    public :: combine_polarization_states

(Polarizations: procedures)+≡
    subroutine combine_polarization_states (pol, state)
        type(polarization_t), dimension(:), intent(in), target :: pol
        type(state_matrix_t), intent(out) :: state
        call outer_multiply (pol%state, state)
    end subroutine combine_polarization_states

```

Transform a polarization density matrix into a polarization vector. This is possible without information loss only for spin-1/2 and for massless particles. To get a unique answer in all cases, we consider only the components with highest weight. Obviously, this loses the longitudinal component of a massive vector, for instance.

This is the inverse operation of `polarization_init_axis` above, where the polarization fraction is set to unity.

```

(Polarizations: public)+≡
    public :: polarization_get_axis

(Polarizations: procedures)+≡
    function polarization_get_axis (pol) result (alpha)
        real(default), dimension(3) :: alpha
        type(polarization_t), intent(in), target :: pol
        type(state_iterator_t) :: it
        complex(default), dimension(2,2) :: value
        type(helicity_t), dimension(2,2) :: hel
        type(helicity_t), dimension(1) :: hel1
        integer :: hmax, i, j
        if (pol%polarized) then
            hmax = pol%spin_type / 2
            call helicity_init (hel(1,1), hmax, hmax)

```

```

call helicity_init (hel(1,2), hmax,-hmax)
call helicity_init (hel(2,1),-hmax, hmax)
call helicity_init (hel(2,2),-hmax,-hmax)
value = 0
call state_iterator_init (it, pol%state)
do while (state_iterator_is_valid (it))
  hel1 = state_iterator_get_helicity (it)
  SCAN_HEL: do i = 1, 2
    do j = 1, 2
      if (hel1(1) == hel(i,j)) then
        value(i,j) = state_iterator_get_matrix_element (it)
        exit SCAN_HEL
      end if
    end do
  end do SCAN_HEL
  call state_iterator_advance (it)
end do
alpha(1) = value(1,2) + value(2,1)
alpha(2) = imago * (value(1,2) - value(2,1))
alpha(3) = value(1,1) - value(2,2)
else
  alpha = 0
end if
end function polarization_get_axis

```

This function returns polarization degree and polar and azimuthal angles ( $\theta, \phi$ ) of the polarization axis.

*(Polarizations: public)*+≡

public :: polarization\_to\_angles

*(Polarizations: procedures)*+≡

```

subroutine polarization_to_angles (pol, r, theta, phi)
  type(polarization_t), intent(in) :: pol
  real(default), intent(out) :: r, theta, phi
  real(default), dimension(3) :: alpha
  real(default) :: r12
  if (pol%polarized) then
    alpha = polarization_get_axis (pol)
    r = sqrt (sum (alpha**2))
    if (any (alpha /= 0)) then
      r12 = sqrt (alpha(1)**2 + alpha(2)**2)
      theta = atan2 (r12, alpha(3))
      if (any (alpha(1:2) /= 0)) then
        phi = atan2 (alpha(2), alpha(1))
      else
        phi = 0
      end if
    else
      theta = 0
    end if
  else
    r = 0
    theta = 0
    phi = 0
  end if
end subroutine polarization_to_angles

```

```

end if
end subroutine polarization_to_angles

```

### 8.1.8 Test

```

<Polarizations: public>+≡
  public :: polarization_test

<Polarizations: procedures>+≡
  subroutine polarization_test
    use os_interface, only: os_data_t
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(polarization_t) :: pol
    type(flavor_t) :: flv
    real(default), dimension(3) :: alpha
    real(default) :: r, theta, phi
    print *, "* Read model file"
    call syntax_model_file_init ()
    call model_list_read_model &
      (var_str("SM"), var_str("SM.mdl"), os_data, model)
    print *, "Unpolarized fermion"
    call flavor_init (flv, 1, model)
    call polarization_init_unpolarized (pol, flv)
    call polarization_write (pol)
    print *, "diagonal =", polarization_is_diagonal (pol)
    call polarization_final (pol)
    print *, "Unpolarized fermion"
    call polarization_init_circular (pol, flv, 0._default)
    call polarization_write (pol)
    call polarization_final (pol)
    print *, "Transversally polarized fermion, phi=0"
    call polarization_init_transversal (pol, flv, 0._default, 1._default)
    call polarization_write (pol)
    print *, "diagonal =", polarization_is_diagonal (pol)
    call polarization_final (pol)
    print *, "Transversally polarized fermion, phi=0.9, frac=0.8"
    call polarization_init_transversal (pol, flv, 0.9_default, 0.8_default)
    call polarization_write (pol)
    print *, "diagonal =", polarization_is_diagonal (pol)
    call polarization_final (pol)
    print *, "All polarization directions of a fermion"
    call polarization_init_generic (pol, flv)
    call polarization_write (pol)
    call polarization_final (pol)
    call flavor_init (flv, 21, model)
    print *, "Circularly polarized gluon, frac=0.3"
    call polarization_init_circular (pol, flv, 0.3_default)
    call polarization_write (pol)
    call polarization_final (pol)
    call flavor_init (flv, 23, model)
    print *, "Circularly polarized massive vector, frac=-0.7"
    call polarization_init_circular (pol, flv, -0.7_default)

```

```

call polarization_write (pol)
call polarization_final (pol)
print *, "Circularly polarized massive vector"
call polarization_init_circular (pol, flv, 1._default)
call polarization_write (pol)
call polarization_final (pol)
print *, "Longitudinally polarized massive vector, frac=0.4"
call polarization_init_longitudinal (pol, flv, 0.4_default)
call polarization_write (pol)
call polarization_final (pol)
print *, "Longitudinally polarized massive vector"
call polarization_init_longitudinal (pol, flv, 1._default)
call polarization_write (pol)
call polarization_final (pol)
print *, "Diagonally polarized massive vector"
call polarization_init_diagonal &
    (pol, flv, (/0._default, 1._default, 2._default/))
call polarization_write (pol)
call polarization_final (pol)
print *, "All polarization directions of a massive vector"
call polarization_init_generic (pol, flv)
call polarization_write (pol)
call polarization_final (pol)
call flavor_init (flv, 21, model)
print *, "Axis polarization (0.2, 0.4, 0.6)"
alpha = (/0.2_default, 0.4_default, 0.6_default/)
call polarization_init_axis (pol, flv, alpha)
call polarization_write (pol)
print *, "Recovered axis:"
alpha = polarization_get_axis (pol)
print *, "Angle polarization (0.5, 0.6, -1)"
r = 0.5_default
theta = 0.6_default
phi = -1._default
call polarization_init_angles (pol, flv, r, theta, phi)
call polarization_write (pol)
print *, "Recovered parameters (r, theta, phi):"
call polarization_to_angles (pol, r, theta, phi)
print *, r, theta, phi
call polarization_final (pol)
end subroutine polarization_test

```

## 8.2 Particles

This module defines the `particle_t` object type, and the methods and operations that deal with it.

```

<particles.f90>≡
  <File header>

```

```

module particles

```



```

<Use kinds>
<Use strings>
<Use file utils>
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use subevents
  use expressions
  use models
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use interactions
  use evaluators
  use polarizations
!  use event_formats
!  use hepmc_interface

<Standard module head>

<Particles: public>

<Particles: parameters>

<Particles: types>

<Particles: interfaces>

contains

<Particles: procedures>

end module particles

```

### 8.2.1 The particle type

#### Particle status codes

The overall status codes (incoming/outgoing etc.) are inherited from the module `subevents`.

Polarization status:

```

<Particles: parameters>≡
  integer, parameter, public :: PRT_UNPOLARIZED = 0
  integer, parameter, public :: PRT_DEFINITE_HELICITY = 1
  integer, parameter, public :: PRT_GENERIC_POLARIZATION = 2

```

#### Definition

The quantum numbers are flavor (from which invariant particle properties can be derived), color, and polarization. The particle may be unpolarized. In this case, `hel` and `pol` are unspecified. If it has a definite helicity, the `hel` component

is defined. If it has a generic polarization, the `pol` component is defined. For each particle we store the four-momentum and the invariant mass squared, i.e., the squared norm of the four-momentum. There is also an optional list of parent and child particles, for bookkeeping in physical events.

```

(Particles: public)≡
    public :: particle_t

(Particles: types)≡
    type :: particle_t
    private
        integer :: status = PRT_UNDEFINED
        integer :: polarization = PRT_UNPOLARIZED
        type(flavor_t) :: flv
        type(color_t) :: col
        type(helicity_t) :: hel
        type(polarization_t) :: pol
        type(vector4_t) :: p = vector4_null
        real(default) :: p2 = 0
        integer, dimension(:), allocatable :: parent
        integer, dimension(:), allocatable :: child
    end type particle_t

```

Particle initializers:

```

(Particles: interfaces)≡
    interface particle_init
        module procedure particle_init_particle
        module procedure particle_init_state
    !     module procedure particle_init_hepmc
    end interface

```

Copy a particle. (Deep copy) The excludes the parent-child relations.

```

(Particles: procedures)≡
    subroutine particle_init_particle (prt_out, prt_in)
        type(particle_t), intent(out) :: prt_out
        type(particle_t), intent(in) :: prt_in
        prt_out%status = prt_in%status
        prt_out%polarization = prt_in%polarization
        prt_out%flv = prt_in%flv
        prt_out%col = prt_in%col
        prt_out%hel = prt_in%hel
        prt_out%pol = prt_in%pol
        prt_out%p = prt_in%p
        prt_out%p2 = prt_in%p2
    end subroutine particle_init_particle

```

Initialize a particle using a single-particle state matrix which determines flavor, color, and polarization. The state matrix must have unique flavor and color. The factorization mode determines whether the particle is unpolarized, has definite helicity, or generic polarization. This mode is translated into the polarization status.

```

(Particles: procedures)+≡
    subroutine particle_init_state (prt, state, status, mode)

```

```

type(particle_t), intent(out) :: prt
type(state_matrix_t), intent(in) :: state
integer, intent(in) :: status, mode
type(state_iterator_t) :: it
prt%status = status
call state_iterator_init (it, state)
prt%flv = state_iterator_get_flavor (it, 1)
if (flavor_is_beam_remnant (prt%flv)) prt%status = PRT_BEAM_REMNANT
prt%col = state_iterator_get_color (it, 1)
select case (mode)
case (FM_SELECT_HELICITY)
    prt%hel = state_iterator_get_helicity (it, 1)
    prt%polarization = PRT_DEFINITE_HELICITY
case (FM_FACTOR_HELICITY)
    call polarization_init_state_matrix (prt%pol, state)
    prt%polarization = PRT_GENERIC_POLARIZATION
end select
end subroutine particle_init_state

```

Initialize a particle from a HepMC particle object. The model is necessary for making a fully qualified flavor component. We have the additional flag `polarized` which tells whether the polarization information should be interpreted or ignored, and the lookup array of barcodes. Note that the lookup array is searched linearly, a possible bottleneck for large particle arrays. If necessary, the barcode array could be replaced by a hash table.

*(CCC Particles: procedures)*≡

```

subroutine particle_init_hepmc (prt, hpert, model, polarization, barcode)
    type(particle_t), intent(out) :: prt
    type(hepmc_particle_t), intent(in) :: hpert
    type(model_t), intent(in), target :: model
    integer, intent(in) :: polarization
    integer, dimension(:), intent(in) :: barcode
    type(hepmc_polarization_t) :: hpol
    integer :: n_parents, n_children
    integer, dimension(:), allocatable :: parent_barcode, child_barcode
    integer :: i
    select case (hepmc_particle_get_status (hpert))
    case (1); prt%status = PRT_OUTGOING
    case (2); prt%status = PRT_RESONANT
    case (3); prt%status = PRT_VIRTUAL
    end select
    call flavor_init (prt%flv, hepmt_particle_get_pdg (hpert), model)
    if (flavor_is_beam_remnant (prt%flv)) prt%status = PRT_BEAM_REMNANT
    call color_init (prt%col, hepmt_particle_get_color (hpert))
    prt%polarization = polarization
    select case (polarization)
    case (PRT_DEFINITE_HELICITY)
        hpol = hepmt_particle_get_polarization (hpert)
        call hepmt_polarization_to_hel (hpol, prt%flv, prt%hel)
        call hepmt_polarization_final (hpol)
    case (PRT_GENERIC_POLARIZATION)
        hpol = hepmt_particle_get_polarization (hpert)
        call hepmt_polarization_to_pol (hpol, prt%flv, prt%pol)
    end select
end subroutine particle_init_hepmc

```

```

        call hepmc_polarization_final (hpol)
    end select
    prt%p = hepmc_particle_get_momentum (hprt)
    prt%p2 = hepmc_particle_get_mass_squared (hprt)
    n_parents = hepmc_particle_get_n_parents (hprt)
    n_children = hepmc_particle_get_n_children (hprt)
    allocate (parent_barcode (n_parents), prt%parent (n_parents))
    allocate (child_barcode (n_children), prt%child (n_children))
    parent_barcode = hepmc_particle_get_parent_barcodes (hprt)
    child_barcode = hepmc_particle_get_child_barcodes (hprt)
    do i = 1, size (barcode)
        where (parent_barcode == barcode(i)) prt%parent = i
        where (child_barcode == barcode(i)) prt%child = i
    end do
    if (prt%status == PRT_VIRTUAL .and. n_parents == 0) &
        prt%status = PRT_INCOMING
end subroutine particle_init_hepmc

```

Finalizer. The polarization component has pointers allocated.

```

<Particles: procedures>+≡
    elemental subroutine particle_final (prt)
        type(particle_t), intent(inout) :: prt
        call polarization_final (prt%pol)
    end subroutine particle_final

```

## I/O

```

<Particles: public>+≡
    public :: particle_write

<Particles: procedures>+≡
    subroutine particle_write (prt, unit)
        type(particle_t), intent(in) :: prt
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return
        select case (prt%status)
            case (PRT_UNDEFINED); write (u, "(1x, A)", advance="no") "[-]"
            case (PRT_BEAM); write (u, "(1x, A)", advance="no") "[b]"
            case (PRT_INCOMING); write (u, "(1x, A)", advance="no") "[i]"
            case (PRT_OUTGOING); write (u, "(1x, A)", advance="no") "[o]"
            case (PRT_VIRTUAL); write (u, "(1x, A)", advance="no") "[v]"
            case (PRT_RESONANT); write (u, "(1x, A)", advance="no") "[r]"
            case (PRT_BEAM_REMNANT); write (u, "(1x, A)", advance="no") "[x]"
        end select
        write (u, "(1x)", advance="no")
        call flavor_write (prt%flv, unit)
        call color_write (prt%col, unit)
        select case (prt%polarization)
            case (PRT_DEFINITE_HELICITY)
                call helicity_write (prt%hel, unit)
                write (u, *)
            case (PRT_GENERIC_POLARIZATION)

```

```

        write (u, *)
        call polarization_write (prt%pol, unit)
    case default
        write (u, *)
    end select
    call vector4_write (prt%p, unit)
    write (u, "(1x,A,1x,ES19.12)") "T = ", prt%p2
    if (allocated (prt%parent)) then
        if (size (prt%parent) /= 0) then
            write (u, "(1x,A,40(1x,I0))") "Parents: ", prt%parent
        end if
    end if
    if (allocated (prt%child)) then
        if (size (prt%child) /= 0) then
            write (u, "(1x,A,40(1x,I0))") "Children:", prt%child
        end if
    end if
end subroutine particle_write

```

Binary I/O:

*(Particles: procedures)*+≡

```

subroutine particle_write_raw (prt, u)
    type(particle_t), intent(in) :: prt
    integer, intent(in) :: u
    write (u) prt%status, prt%polarization
    call flavor_write_raw (prt%flv, u)
    call color_write_raw (prt%col, u)
    select case (prt%polarization)
    case (PRT_DEFINITE_HELICITY)
        call helicity_write_raw (prt%hel, u)
    case (PRT_GENERIC_POLARIZATION)
        call polarization_write_raw (prt%pol, u)
    end select
    call vector4_write_raw (prt%p, u)
    write (u) prt%p2
    write (u) allocated (prt%parent)
    if (allocated (prt%parent)) then
        write (u) size (prt%parent)
        write (u) prt%parent
    end if
    write (u) allocated (prt%child)
    if (allocated (prt%child)) then
        write (u) size (prt%child)
        write (u) prt%child
    end if
end subroutine particle_write_raw

subroutine particle_read_raw (prt, u, iostat)
    type(particle_t), intent(out) :: prt
    integer, intent(in) :: u
    integer, intent(out) :: iostat
    logical :: allocated_parent, allocated_child
    integer :: size_parent, size_child

```

```

read (u, iostat=iostat) prt%status, prt%polarization
call flavor_read_raw (prt%flv, u, iostat=iostat)
call color_read_raw (prt%col, u, iostat=iostat)
select case (prt%polarization)
case (PRT_DEFINITE_HELICITY)
    call helicity_read_raw (prt%hel, u, iostat=iostat)
case (PRT_GENERIC_POLARIZATION)
    call polarization_read_raw (prt%pol, u, iostat=iostat)
end select
call vector4_read_raw (prt%p, u, iostat=iostat)
read (u, iostat=iostat) prt%p2
read (u, iostat=iostat) allocated_parent
if (allocated_parent) then
    read (u, iostat=iostat) size_parent
    allocate (prt%parent (size_parent))
    read (u, iostat=iostat) prt%parent
end if
read (u, iostat=iostat) allocated_child
if (allocated_child) then
    read (u, iostat=iostat) size_child
    allocate (prt%child (size_child))
    read (u, iostat=iostat) prt%child
end if
end subroutine particle_read_raw

```

## Setting contents

Reset the status code.

```

<Particles: public>+≡
    public :: particle_reset_status

<Particles: procedures>+≡
    elemental subroutine particle_reset_status (prt, status)
        type(particle_t), intent(inout) :: prt
        integer, intent(in) :: status
        prt%status = status
    end subroutine particle_reset_status

```

The color can be given explicitly.

```

<Particles: public>+≡
    public :: particle_set_color

<Particles: procedures>+≡
    elemental subroutine particle_set_color (prt, col)
        type(particle_t), intent(inout) :: prt
        type(color_t), intent(in) :: col
        prt%col = col
    end subroutine particle_set_color

```

The flavor can be given explicitly.

```

<Particles: public>+≡
    public :: particle_set_flavor

```

```

<Particles: procedures>+≡
  subroutine particle_set_flavor (prt, flv)
    type(particle_t), intent(inout) :: prt
    type(flavor_t), intent(in) :: flv
    prt%flv = flv
  end subroutine particle_set_flavor

```

Manually set the model for the particle flavor. This is required, e.g., if the particle has been read from file.

```

<Particles: public>+≡
  public :: particle_set_model

<Particles: procedures>+≡
  subroutine particle_set_model (prt, model)
    type(particle_t), intent(inout) :: prt
    type(model_t), intent(in), target :: model
    call flavor_set_model (prt%flv, model)
  end subroutine particle_set_model

```

The momentum is set independent of the quantum numbers.

```

<Particles: public>+≡
  public :: particle_set_momentum

<Particles: procedures>+≡
  elemental subroutine particle_set_momentum (prt, p, on_shell)
    type(particle_t), intent(inout) :: prt
    type(vector4_t), intent(in) :: p
    logical, intent(in), optional :: on_shell
    prt%p = p
    if (present (on_shell)) then
      if (on_shell) then
        if (flavor_is_associated (prt%flv)) then
          prt%p2 = flavor_get_mass (prt%flv) ** 2
          return
        end if
      end if
    end if
    prt%p2 = p ** 2
  end subroutine particle_set_momentum

```

Set resonance information. This should be done after momentum assignment, because we need to know whether the particle is spacelike or timelike. The resonance flag is defined only for virtual particles.

```

<Particles: procedures>+≡
  elemental subroutine particle_set_resonance_flag (prt, resonant)
    type(particle_t), intent(inout) :: prt
    logical, intent(in) :: resonant
    select case (prt%status)
    case (PRT_VIRTUAL)
      if (resonant) prt%status = PRT_RESONANT
    end select
  end subroutine particle_set_resonance_flag

```

Set children and parents information.

```

(Particles: public) +=
  public :: particle_set_children
  public :: particle_set_parents

(Particles: procedures) +=
  subroutine particle_set_children (prt, idx)
    type(particle_t), intent(inout) :: prt
    integer, dimension(:), intent(in) :: idx
    if (allocated (prt%child)) deallocate (prt%child)
    allocate (prt%child (count (idx /= 0)))
    prt%child = pack (idx, idx /= 0)
  end subroutine particle_set_children

  subroutine particle_set_parents (prt, idx)
    type(particle_t), intent(inout) :: prt
    integer, dimension(:), intent(in) :: idx
    if (allocated (prt%parent)) deallocate (prt%parent)
    allocate (prt%parent (count (idx /= 0)))
    prt%parent = pack (idx, idx /= 0)
  end subroutine particle_set_parents

```

## Accessing contents

The status code.

```

(Particles: public) +=
  public :: particle_get_status

(Particles: procedures) +=
  elemental function particle_get_status (prt) result (status)
    integer :: status
    type(particle_t), intent(in) :: prt
    status = prt%status
  end function particle_get_status

```

Return true if the status is either INCOMING, OUTGOING or RESONANT. BEAM is kept, if keep\_beams is set true.

```

(Particles: procedures) +=
  elemental function particle_is_real (prt, keep_beams) result (flag)
    logical :: flag, kb
    type(particle_t), intent(in) :: prt
    logical, intent(in), optional :: keep_beams
    kb = .false.
    if (present (keep_beams)) kb = keep_beams
    select case (prt%status)
    case (PRT_INCOMING, PRT_OUTGOING, PRT_RESONANT)
      flag = .true.
    case (PRT_BEAM)
      flag = kb
    case default
      flag = .false.
    end select

```



```
end function particle_is_real
```

Polarization status.

```
<Particles: public>+≡
  public :: particle_get_polarization_status

<Particles: procedures>+≡
  elemental function particle_get_polarization_status (prt) result (status)
    integer :: status
    type(particle_t), intent(in) :: prt
    status = prt%polarization
  end function particle_get_polarization_status
```

Return the PDG code from the flavor component directly.

```
<Particles: public>+≡
  public :: particle_get_pdg

<Particles: procedures>+≡
  elemental function particle_get_pdg (prt) result (pdg)
    integer :: pdg
    type(particle_t), intent(in) :: prt
    pdg = flavor_get_pdg (prt%flv)
  end function particle_get_pdg
```

Return the color and anticolor quantum numbers.

```
<Particles: public>+≡
  public :: particle_get_color

<Particles: procedures>+≡
  function particle_get_color (prt) result (col)
    integer, dimension(2) :: col
    type(particle_t), intent(in) :: prt
    col(1) = color_get_col (prt%col)
    col(2) = color_get_acl (prt%col)
  end function particle_get_color
```

Return the polarization density matrix (as a shallow copy).

```
<Particles: public>+≡
  public :: particle_get_polarization

<Particles: procedures>+≡
  function particle_get_polarization (prt) result (pol)
    type(polarization_t) :: pol
    type(particle_t), intent(in) :: prt
    pol = prt%pol
  end function particle_get_polarization
```

Return the helicity (if defined and diagonal).

```
<Particles: public>+≡
  public :: particle_get_helicity
```

```

(Particles: procedures) +=
function particle_get_helicity (prt) result (hel)
    integer :: hel
    integer, dimension(2) :: hel_arr
    type(particle_t), intent(in) :: prt
    hel = 0
    if (helicity_is_defined (prt%hel) .and. &
        helicity_is_diagonal (prt%hel)) then
        hel_arr = helicity_get (prt%hel)
        hel = hel_arr (1)
    end if
end function particle_get_helicity

```

Return the number of children/parents

```

(Particles: public) +=
public :: particle_get_n_parents
public :: particle_get_n_children

(Particles: procedures) +=
function particle_get_n_parents (prt) result (n)
    integer :: n
    type(particle_t), intent(in) :: prt
    if (allocated (prt%parent)) then
        n = size (prt%parent)
    else
        n = 0
    end if
end function particle_get_n_parents

function particle_get_n_children (prt) result (n)
    integer :: n
    type(particle_t), intent(in) :: prt
    if (allocated (prt%child)) then
        n = size (prt%child)
    else
        n = 0
    end if
end function particle_get_n_children

```

Return the array of parents/children.

```

(Particles: public) +=
public :: particle_get_parents
public :: particle_get_children

(Particles: procedures) +=
function particle_get_parents (prt) result (parent)
    type(particle_t), intent(in) :: prt
    integer, dimension(:), allocatable :: parent
    if (allocated (prt%parent)) then
        allocate (parent (size (prt%parent)))
        parent = prt%parent
    else
        allocate (parent (0))
    end if
end function particle_get_parents

```

```

end function particle_get_parents

function particle_get_children (prt) result (child)
  type(particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: child
  if (allocated (prt%child)) then
    allocate (child (size (prt%child)))
    child = prt%child
  else
    allocate (child (0))
  end if
end function particle_get_children

```

Return momentum and momentum squared.

```

(Particles: public) +=
  public :: particle_get_momentum
  public :: particle_get_p2

(Particles: procedures) +=
  function particle_get_momentum (prt) result (p)
    type(vector4_t) :: p
    type(particle_t), intent(in) :: prt
    p = prt%p
  end function particle_get_momentum

  function particle_get_p2 (prt) result (p2)
    real(default) :: p2
    type(particle_t), intent(in) :: prt
    p2 = prt%p2
  end function particle_get_p2

```

### 8.2.2 Particle sets

A particle set is what is usually called an event: an array of particles. The individual particle entries carry momentum, quantum numbers, polarization, and optionally connections. There is (also optionally) a correlated state-density matrix that maintains spin correlations that are lost in the individual particle entries.

```

(Particles: public) +=
  public :: particle_set_t

(Particles: types) +=
  type :: particle_set_t
  !   private
  integer :: n_beam = 0
  integer :: n_in = 0
  integer :: n_vir = 0
  integer :: n_out = 0
  integer :: n_tot = 0
  type(particle_t), dimension(:), allocatable :: prt
  type(state_matrix_t) :: correlated_state
end type particle_set_t

```

A particle set can be initialized from an interaction or from a HepMC event record.

```

(Particles: public)+≡
    public :: particle_set_init
(Particles: interfaces)+≡
    interface particle_set_init
        module procedure particle_set_init_interaction
    !     module procedure particle_set_init_hepmc
    end interface

```

When a particle set is initialized from a given interaction, we have to determine the branch within the original state matrix that fixes the particle quantum numbers. This is done with the appropriate probabilities, based on a random number `x`. The `mode` determines whether the individual particles become unpolarized, or take a definite (diagonal) helicity, or acquire single-particle polarization matrices. The flag `keep_correlations` tells whether the spin-correlation matrix is to be calculated and stored in addition to the particles. The flag `keep_virtual` tells whether virtual particles should be dropped. Note that if virtual particles are dropped, the spin-correlation matrix makes no sense, and parent-child relations are not set.

For a correct disentangling of color and flavor (in the presence of helicity), we consider two interactions. `int` has no color information, and is used to select a flavor state. Consequently, we trace over helicities here. `int_flows` contains color-flow and potentially helicity information, but is useful only after the flavor combination has been chosen. So this interaction is used to select helicity and color, but restricted to the selected flavor combination.

`int` and `int_flows` may be identical if there is only a single (or no) color flow. If there is just a single flavor combination, `x(1)` can be set to zero.

The current algorithm of evaluator convolution requires that the beam particles are assumed outgoing (in the beam interaction) and become virtual in all derived interactions. In the particle set they should be re-identified as incoming. The optional integer `n_incoming` can be used to perform this correction.

The flag `is_valid` is false if factorization of the state is not possible, in particular if the squared matrix element is zero.

```

(Particles: procedures)+≡
    subroutine particle_set_init_interaction &
        (particle_set, is_valid, int, int_flows, mode, x, &
         keep_correlations, keep_virtual, n_incoming)
    type(particle_set_t), intent(out) :: particle_set
    logical, intent(out) :: is_valid
    type(interaction_t), intent(in), target :: int, int_flows
    integer, intent(in) :: mode
    real(default), dimension(2), intent(in) :: x
    logical, intent(in) :: keep_correlations, keep_virtual
    integer, intent(in), optional :: n_incoming
    type(state_matrix_t), dimension(:), allocatable, target :: flavor_state
    type(state_matrix_t), dimension(:), allocatable, target :: single_state
    integer :: n_in, n_vir, n_out, n_tot
    type(quantum_numbers_t), dimension(:,:), allocatable :: qn
    logical :: ok
    integer :: i, j

```

```

if (present (n_incoming)) then
  n_in = n_incoming
  n_vir = interaction_get_n_vir (int) - n_incoming
else
  n_in = interaction_get_n_in (int)
  n_vir = interaction_get_n_vir (int)
end if
n_out = interaction_get_n_out (int)
n_tot = interaction_get_n_tot (int)
particle_set%n_in = n_in
particle_set%n_out = n_out
if (keep_virtual) then
  particle_set%n_vir = n_vir
  particle_set%n_tot = n_tot
else
  particle_set%n_vir = 0
  particle_set%n_tot = n_in + n_out
end if
call interaction_factorize &
  (int, FM_IGNORE_HELICITY, x(1), is_valid, flavor_state)
allocate (qn (n_tot,1))
do i = 1, n_tot
  qn(i,:) = state_matrix_get_quantum_numbers (flavor_state(i), 1)
end do
if (keep_correlations .and. keep_virtual) then
  call interaction_factorize (int_flows, mode, x(2), ok, &
    single_state, particle_set%correlated_state, qn(:,1))
else
  call interaction_factorize (int_flows, mode, x(2), ok, &
    single_state, qn_in=qn(:,1))
end if
is_valid = is_valid .and. ok
allocate (particle_set%prt (particle_set%n_tot))
j = 1
do i = 1, n_tot
  if (i <= n_in) then
    call particle_init &
      (particle_set%prt(j), single_state(i), PRT_INCOMING, mode)
    call particle_set_momentum &
      (particle_set%prt(j), interaction_get_momentum (int, i))
  else if (i <= n_in + n_vir) then
    if (.not. keep_virtual) cycle
    call particle_init &
      (particle_set%prt(j), single_state(i), PRT_VIRTUAL, mode)
    call particle_set_momentum &
      (particle_set%prt(j), interaction_get_momentum (int, i))
  else
    call particle_init &
      (particle_set%prt(j), single_state(i), PRT_OUTGOING, mode)
    call particle_set_momentum &
      (particle_set%prt(j), interaction_get_momentum (int, i), &
        on_shell = .true.)
  end if
  if (keep_virtual) then

```

```

        call particle_set_children &
            (particle_set%prt(j), interaction_get_children (int, i))
        call particle_set_parents &
            (particle_set%prt(j), interaction_get_parents (int, i))
    end if
    j = j + 1
end do
if (keep_virtual) then
    call particle_set_resonance_flag &
        (particle_set%prt, interaction_get_resonance_flags (int))
end if
call state_matrix_final (flavor_state)
call state_matrix_final (single_state)
end subroutine particle_set_init_interaction

```

If a particle set is initialized from a HepMC event record, we have to specify the treatment of polarization (unpolarized or density matrix) which is common to all particles. Correlated polarization information is not available.

*(CCC Particles: procedures)*+≡

```

subroutine particle_set_init_hepmc (particle_set, evt, model, polarization)
    type(particle_set_t), intent(out) :: particle_set
    type(hepmc_event_t), intent(in) :: evt
    type(model_t), intent(in), target :: model
    integer, intent(in) :: polarization
    type(hepmc_event_particle_iterator_t) :: it
    type(hepmc_particle_t) :: prt
    integer, dimension(:), allocatable :: barcode
    integer :: n_tot, i
    n_tot = 0
    call hepmc_event_particle_iterator_init (it, evt)
    do while (hepmc_event_particle_iterator_is_valid (it))
        n_tot = n_tot + 1
        call hepmc_event_particle_iterator_advance (it)
    end do
    allocate (barcode (n_tot))
    call hepmc_event_particle_iterator_reset (it)
    do i = 1, n_tot
        barcode(i) = hepmc_particle_get_barcode &
            (hepmc_event_particle_iterator_get (it))
        call hepmc_event_particle_iterator_advance (it)
    end do
    allocate (particle_set%prt (n_tot))
    call hepmc_event_particle_iterator_reset (it)
    do i = 1, n_tot
        prt = hepmc_event_particle_iterator_get (it)
        call particle_init (particle_set%prt(i), &
            prt, model, polarization, barcode)
        call hepmc_event_particle_iterator_advance (it)
    end do
    call hepmc_event_particle_iterator_final (it)
    particle_set%n_tot = n_tot
    particle_set%n_beam = &
        count (particle_get_status (particle_set%prt) == PRT_BEAM)

```

```

particle_set%n_in = &
    count (particle_get_status (particle_set%prt) == PRT_INCOMING)
particle_set%n_out = &
    count (particle_get_status (particle_set%prt) == PRT_OUTGOING)
particle_set%n_vir = &
    particle_set%n_tot - particle_set%n_in - particle_set%n_out
end subroutine particle_set_init_hePMC

```

Manually set the model for the stored particles.

```

<Particles: public>+≡
    public :: particle_set_set_model

<Particles: procedures>+≡
    subroutine particle_set_set_model (particle_set, model)
        type(particle_set_t), intent(inout) :: particle_set
        type(model_t), intent(in), target :: model
        integer :: i
        do i = 1, particle_set%n_tot
            call particle_set_model (particle_set%prt(i), model)
        end do
    end subroutine particle_set_set_model

```

Pointer components are hidden inside the particle polarization, and in the correlated state matrix.

```

<Particles: public>+≡
    public :: particle_set_final

<Particles: procedures>+≡
    subroutine particle_set_final (particle_set)
        type(particle_set_t), intent(inout) :: particle_set
        if (allocated (particle_set%prt)) then
            call particle_final (particle_set%prt)
            deallocate(particle_set%prt)
        end if
        call state_matrix_final (particle_set%correlated_state)
    end subroutine particle_set_final

```

Output (default format)

```

<Particles: public>+≡
    public :: particle_set_write

<Particles: procedures>+≡
    subroutine particle_set_write (particle_set, unit)
        type(particle_set_t), intent(in) :: particle_set
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "Particle set:"
        call write_separator (u)
        if (particle_set%n_tot /= 0) then
            do i = 1, particle_set%n_tot
                write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
                call particle_write (particle_set%prt(i), u)
            end do
        end if
    end subroutine particle_set_write

```

```

        if (state_matrix_is_defined (particle_set%correlated_state)) then
            call write_separator (u)
            write (u, *) "Correlated state density matrix:"
            call state_matrix_write (particle_set%correlated_state, u)
        end if
    else
        write (u, "(3x,A)") "[empty]"
    end if
end subroutine particle_set_write

```

### 8.2.3 I/O formats

Here, we define input/output of particle sets in various formats. This is the right place since particle sets contain most of the event information.

All write/read routines take as first argument the object, as second argument the I/O unit which in this case is a mandatory argument. Then follow further event data.

#### Internal binary format

This format is supposed to contain the complete information, so the particle data set can be fully reconstructed. The exception is the model part of the particle flavors; this is unassigned for the flavor values read from file.

```

(Particles: public) +=
    public :: particle_set_write_raw
    public :: particle_set_read_raw

(Particles: procedures) +=
    subroutine particle_set_write_raw (particle_set, u)
        type(particle_set_t), intent(in) :: particle_set
        integer, intent(in) :: u
        integer :: i
        write (u) &
            particle_set%n_beam, particle_set%n_in, &
            particle_set%n_vir, particle_set%n_out
        write (u) particle_set%n_tot
        do i = 1, particle_set%n_tot
            call particle_write_raw (particle_set%prt(i), u)
        end do
        call state_matrix_write_raw (particle_set%correlated_state, u)
    end subroutine particle_set_write_raw

    subroutine particle_set_read_raw (particle_set, u, iostat)
        type(particle_set_t), intent(out) :: particle_set
        integer, intent(in) :: u
        integer, intent(out) :: iostat
        integer :: i
        read (u, iostat=iostat) &
            particle_set%n_beam, particle_set%n_in, &
            particle_set%n_vir, particle_set%n_out
        read (u, iostat=iostat) particle_set%n_tot
        allocate (particle_set%prt (particle_set%n_tot))
    end subroutine particle_set_read_raw

```



```

do i = 1, size (particle_set%prt)
  call particle_read_raw (particle_set%prt(i), u, iostat=iostat)
end do
call state_matrix_read_raw (particle_set%correlated_state, u, iostat=iostat)
end subroutine particle_set_read_raw

```

## HepMC format

The master output function fills a HepMC GenEvent object that is already initialized, but has no vertices in it.

We first set up the vertex lists and enter the vertices into the HepMC event. Then, we assign first all incoming particles and then all outgoing particles to their associated vertices. Particles which have neither parent nor children entries (this should not happen) are dropped.

Finally, we insert the beam particles. If there are none, use the incoming particles instead.

*<CCC Particles: public>*≡

```
public :: particle_set_fill_hepmc_event
```

*<CCC Particles: procedures>*+≡

```

subroutine particle_set_fill_hepmc_event (particle_set, evt)
  type(particle_set_t), intent(in) :: particle_set
  type(hepmc_event_t), intent(inout) :: evt
  type(hepmc_vertex_t), dimension(:), allocatable :: v
  type(hepmc_particle_t), dimension(:), allocatable :: hp
  type(hepmc_particle_t), dimension(2) :: hbeam
  logical, dimension(:), allocatable :: is_beam
  integer, dimension(:), allocatable :: v_from, v_to
  integer :: n_vertices, n_tot, i
  n_tot = particle_set%n_tot
  allocate (v_from (n_tot), v_to (n_tot))
  call particle_set_assign_vertices (particle_set, v_from, v_to, n_vertices)
  allocate (v (n_vertices))
  do i = 1, n_vertices
    call hepmc_vertex_init (v(i))
    call hepmc_event_add_vertex (evt, v(i))
  end do
  allocate (hp (n_tot))
  do i = 1, n_tot
    if (v_to(i) /= 0 .or. v_from(i) /= 0) then
      call particle_to_hepmc (particle_set%prt(i), hp(i))
    end if
  end do
  allocate (is_beam (n_tot))
  is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_BEAM
  if (.not. any (is_beam)) then
    is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_INCOMING
  end if
  if (count (is_beam) == 2) then
    hbeam = pack (hp, is_beam)
    call hepmc_event_set_beam_particles (evt, hbeam(1), hbeam(2))
  end if

```

```

do i = 1, n_tot
  if (v_to(i) /= 0) then
    call hepvc_vertex_add_particle_in (v(v_to(i)), hpvt(i))
  end if
end do
do i = 1, n_tot
  if (v_from(i) /= 0) then
    call hepvc_vertex_add_particle_out (v(v_from(i)), hpvt(i))
  end if
end do
end subroutine particle_set_fill_hepvc_event

```

### Get contents

Find parents/children of a particular particle recursively; the search terminates if a parent/child has status BEAM, INCOMING, OUTGOING or RESONANT.

*(Particles: procedures)+≡*

```

function particle_set_get_real_parents (pset, i, keep_beams) result (parent)
  integer, dimension(:), allocatable :: parent
  type(particle_set_t), intent(in) :: pset
  integer, intent(in) :: i
  logical, intent(in), optional :: keep_beams
  logical, dimension(:), allocatable :: is_real
  logical, dimension(:), allocatable :: is_parent, is_real_parent
  logical :: kb
  integer :: j, k
  kb = .false.
  if (present (keep_beams)) kb = keep_beams
  allocate (is_real (pset%n_tot))
  is_real = particle_is_real (pset%prt, kb)
  allocate (is_parent (pset%n_tot), is_real_parent (pset%n_tot))
  is_real_parent = .false.
  is_parent = .false.
  is_parent(particle_get_parents(pset%prt(i))) = .true.
  do while (any (is_parent))
    where (is_real .and. is_parent)
      is_real_parent = .true.
      is_parent = .false.
    end where
    mark_next_parent: do j = size (is_parent), 1, -1
      if (is_parent(j)) then
        is_parent(particle_get_parents(pset%prt(j))) = .true.
        is_parent(j) = .false.
        exit mark_next_parent
      end if
    end do mark_next_parent
  end do
  allocate (parent (count (is_real_parent)))
  j = 0
  do k = 1, size (is_parent)
    if (is_real_parent(k)) then
      j = j + 1
    end if
  end do

```

```

        parent(j) = k
    end if
end do
end function particle_set_get_real_parents

function particle_set_get_real_children (pset, i, keep_beams) result (child)
    integer, dimension(:), allocatable :: child
    type(particle_set_t), intent(in) :: pset
    integer, intent(in) :: i
    logical, dimension(:), allocatable :: is_real
    logical, dimension(:), allocatable :: is_child, is_real_child
    logical, intent(in), optional :: keep_beams
    integer :: j, k
    logical :: kb
    kb = .false.
    if (present (keep_beams)) kb = keep_beams
    allocate (is_real (pset%n_tot))
    is_real = particle_is_real (pset%prt, kb)
    allocate (is_child (pset%n_tot), is_real_child (pset%n_tot))
    is_real_child = .false.
    is_child = .false.
    is_child(particle_get_children(pset%prt(i))) = .true.
    do while (any (is_child))
        where (is_real .and. is_child)
            is_real_child = .true.
            is_child = .false.
        end where
        mark_next_child: do j = 1, size (is_child)
            if (is_child(j)) then
                is_child(particle_get_children(pset%prt(j))) = .true.
                is_child(j) = .false.
                exit mark_next_child
            end if
        end do mark_next_child
    end do
    allocate (child (count (is_real_child)))
    j = 0
    do k = 1, size (is_child)
        if (is_real_child(k)) then
            j = j + 1
            child(j) = k
        end if
    end do
end function particle_set_get_real_children

```

Get the `n_tot`, `n_in`, and `n_out` values out of the particle set.

```

⟨Particles: public⟩+≡
    public :: particle_set_get_n_beam
    public :: particle_set_get_n_in
    public :: particle_set_get_n_vir
    public :: particle_set_get_n_out
    public :: particle_set_get_n_tot

⟨Particles: procedures⟩+≡

```

```

function particle_set_get_n_beam (pset) result (n_beam)
  type(particle_set_t), intent(in) :: pset
  integer :: n_beam
  n_beam = pset%n_beam
end function particle_set_get_n_beam

function particle_set_get_n_in (pset) result (n_in)
  type(particle_set_t), intent(in) :: pset
  integer :: n_in
  n_in = pset%n_in
end function particle_set_get_n_in

function particle_set_get_n_vir (pset) result (n_vir)
  type(particle_set_t), intent(in) :: pset
  integer :: n_vir
  n_vir = pset%n_in
end function particle_set_get_n_vir

function particle_set_get_n_out (pset) result (n_out)
  type(particle_set_t), intent(in) :: pset
  integer :: n_out
  n_out = pset%n_out
end function particle_set_get_n_out

function particle_set_get_n_tot (pset) result (n_tot)
  type(particle_set_t), intent(in) :: pset
  integer :: n_tot
  n_tot = pset%n_tot
end function particle_set_get_n_tot

```

Return a pointer to the particle corresponding to the number

```

<Particles: public>+≡
  public :: particle_set_get_particle

<Particles: procedures>+≡
  function particle_set_get_particle(pset, index) result(particle)
    type(particle_set_t), intent(in) :: pset
    integer, intent(in) :: index
    type(particle_t) :: particle

    particle = pset%prt(index)
  end function particle_set_get_particle

```

## Tools

Given a subevent, reset status codes.

```

<Particles: public>+≡
  public :: particle_set_reset_status

<Particles: procedures>+≡
  subroutine particle_set_reset_status (particle_set, index, status)
    type(particle_set_t), intent(inout) :: particle_set
    integer, dimension(:), intent(in) :: index

```

```

integer, intent(in) :: status
integer :: i
if (allocated (particle_set%prt)) then
  do i = 1, size (index)
    call particle_reset_status (particle_set%prt(index(i)), status)
  end do
end if
particle_set%n_beam = &
  count (particle_get_status (particle_set%prt) == PRT_BEAM)
particle_set%n_in = &
  count (particle_get_status (particle_set%prt) == PRT_INCOMING)
particle_set%n_out = &
  count (particle_get_status (particle_set%prt) == PRT_OUTGOING)
particle_set%n_vir = particle_set%n_tot &
  - particle_set%n_beam - particle_set%n_in - particle_set%n_out
end subroutine particle_set_reset_status

```

Reduce a particle set to the essential entries. The entries kept are those with status INCOMING, OUTGOING or RESONANT. BEAM is kept if `keep_beams` is true. Other entries are skipped. The correlated state matrix, if any, is also ignored.

*(Particles: public)*+≡

```
public :: particle_set_reduce
```

*(Particles: procedures)*+≡

```

subroutine particle_set_reduce (pset_in, pset_out, keep_beams)
  type(particle_set_t), intent(in) :: pset_in
  type(particle_set_t), intent(out) :: pset_out
  logical, intent(in), optional :: keep_beams
  integer, dimension(:), allocatable :: status, map
  integer :: i, j
  logical :: kb
  kb = .false.; if (present (keep_beams)) kb = keep_beams
  allocate (status (pset_in%n_tot))
  status = particle_get_status (pset_in%prt)
  if (kb) pset_out%n_beam = count (status == PRT_BEAM)
  pset_out%n_in = count (status == PRT_INCOMING)
  pset_out%n_vir = count (status == PRT_RESONANT)
  pset_out%n_out = count (status == PRT_OUTGOING)
  pset_out%n_tot = &
    pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
  allocate (pset_out%prt (pset_out%n_tot))
  allocate (map (pset_in%n_tot))
  map = 0
  j = 0
  if (kb) call copy_particles (PRT_BEAM)
  call copy_particles (PRT_INCOMING)
  call copy_particles (PRT_RESONANT)
  call copy_particles (PRT_OUTGOING)
  do i = 1, pset_in%n_tot
    if (map(i) == 0) cycle
! triggers nagfor bug!
!       call particle_set_parents (pset_out%prt(map(i)), &
!         map (particle_set_get_real_parents (pset_in, i)))
!       call particle_set_children (pset_out%prt(map(i)), &

```

```

!          map (particle_set_get_real_children (pset_in, i)))
! workaround:
      call particle_set_parents (pset_out%prt(map(i)), &
        particle_set_get_real_parents (pset_in, i, kb))
      call particle_set_parents (pset_out%prt(map(i)), &
        map (pset_out%prt(map(i))%parent))
      call particle_set_children (pset_out%prt(map(i)), &
        particle_set_get_real_children (pset_in, i, kb))
      call particle_set_children (pset_out%prt(map(i)), &
        map (pset_out%prt(map(i))%child))
    end do
contains
  subroutine copy_particles (stat)
    integer, intent(in) :: stat
    integer :: i
    do i = 1, pset_in%n_tot
      if (status(i) == stat) then
        j = j + 1
        map(i) = j
        call particle_init (pset_out%prt(j), pset_in%prt(i))
      end if
    end do
  end subroutine copy_particles
end subroutine particle_set_reduce

```

Remove the beam particles and beam remnants from the particle set if the keep beams flag is false. If keep beams is not given, the beam particles and the beam remnants are removed. The correlated state matrix, if any, is also ignored.

*(Particles: public)+≡*

```
public :: particle_set_apply_keep_beams
```

*(Particles: procedures)+≡*

```

subroutine particle_set_apply_keep_beams (pset_in, pset_out, keep_beams)
  type(particle_set_t), intent(in) :: pset_in
  type(particle_set_t), intent(out) :: pset_out
  logical, intent(in), optional :: keep_beams
  integer, dimension(:), allocatable :: status, map
  integer :: i, j
  logical :: kb
  kb = .false.; if (present (keep_beams)) kb = keep_beams
  allocate (status (pset_in%n_tot))
  status = particle_get_status (pset_in%prt)
  if (kb) pset_out%n_beam = count (status == PRT_BEAM)
  pset_out%n_in = count (status == PRT_INCOMING)
  if (kb) then
    pset_out%n_vir = count (status == PRT_VIRTUAL) + count (status == PRT_RESONANT) &
      + count (status == PRT_BEAM_REMNANT)
  else
    pset_out%n_vir = count (status == PRT_VIRTUAL) + count (status == PRT_RESONANT)
  end if
  pset_out%n_out = count (status == PRT_OUTGOING)
  pset_out%n_tot = &
    pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
  allocate (pset_out%prt (pset_out%n_tot))

```

```

        allocate (map (pset_in%n_tot))
        map = 0
        j = 0
        if (kb) call copy_particles (PRT_BEAM)
        call copy_particles (PRT_INCOMING)
        if (kb) call copy_particles (PRT_BEAM_REMNANT)
        call copy_particles (PRT_RESONANT)
        call copy_particles (PRT_VIRTUAL)
        call copy_particles (PRT_OUTGOING)
        do i = 1, pset_in%n_tot
            if (map(i) == 0) cycle
! triggers nagfor bug!
!         call particle_set_parents (pset_out%prt(map(i)), &
!             map (particle_set_get_real_parents (pset_in, i)))
!         call particle_set_children (pset_out%prt(map(i)), &
!             map (particle_set_get_real_children (pset_in, i)))
! workaround:
            call particle_set_parents (pset_out%prt(map(i)), &
                particle_set_get_real_parents (pset_in, i, kb))
            call particle_set_parents (pset_out%prt(map(i)), &
                map (pset_out%prt(map(i))%parent))
            call particle_set_children (pset_out%prt(map(i)), &
                particle_set_get_real_children (pset_in, i, kb))
            call particle_set_children (pset_out%prt(map(i)), &
                map (pset_out%prt(map(i))%child))
        end do
contains
        subroutine copy_particles (stat)
            integer, intent(in) :: stat
            integer :: i
            do i = 1, pset_in%n_tot
                if (status(i) == stat) then
                    j = j + 1
                    map(i) = j
                    call particle_init (pset_out%prt(j), pset_in%prt(i))
                end if
            end do
        end subroutine copy_particles
    end subroutine particle_set_apply_keep_beams

```

Transform a particle set into HEPEVT-compatible form. In this form, for each particle, the parents and the children are contiguous in the particle array. Usually, this requires to clone some particles.

We do not know in advance how many particles the canonical form will have. To be on the safe side, allocate four times the original size.

```

<Particles: public>+≡
    public :: particle_set_to_hepevt_form

<Particles: procedures>+≡
    subroutine particle_set_to_hepevt_form (pset_in, pset_out, keep_beams)
        type(particle_set_t), intent(in) :: pset_in
        type(particle_set_t), intent(out) :: pset_out
        logical, intent(in), optional :: keep_beams
        type(particle_set_t) :: pset

```

```

type :: particle_entry_t
  integer :: src = 0
  integer :: status = 0
  integer :: orig = 0
  integer :: copy = 0
end type particle_entry_t
type(particle_entry_t), dimension(:), allocatable :: prt
integer, dimension(:), allocatable :: map1, map2
integer, dimension(:), allocatable :: parent, child
integer :: n_tot, n_parents, n_children, i, j, c, n

call particle_set_apply_keep_beams(pset_in, pset, keep_beams)
n_tot = pset%n_tot
allocate (prt (4 * n_tot))
allocate (map1(4 * n_tot))
allocate (map2(4 * n_tot))
map1 = 0
map2 = 0
allocate (child (n_tot))
allocate (parent (n_tot))
n = 0
do i = 1, n_tot
  if (particle_get_n_parents (pset%prt(i)) == 0) then
    call append (i)
  end if
end do
do i = 1, n_tot
  n_children = particle_get_n_children (pset%prt(i))
  if (n_children > 0) then
    child(1:n_children) = particle_get_children (pset%prt(i))
    c = child(1)
    if (map1(c) == 0) then
      n_parents = particle_get_n_parents (pset%prt(c))
      if (n_parents > 1) then
        parent(1:n_parents) = particle_get_parents (pset%prt(c))
        if (i == parent(1) .and. &
          any( (/map1(i)+j-1, j=1,n_parents)/) /= map1(parent(1:n_parents)))) &
          then
          do j = 1, n_parents
            call append (parent(j))
          end do
        end if
      else if (map1(i) == 0) then
        call append (i)
      end if
      do j = 1, n_children
        call append (child(j))
      end do
    end if
  else if (map1(i) == 0) then
    call append (i)
  end if
end do
do i = n, 1, -1

```



```

        if (prt(i)%status /= PRT_OUTGOING) then
            do j = 1, i-1
                if (prt(j)%status == PRT_OUTGOING) then
                    call append(prt(j)%src)
                end if
            end do
            exit
        end if
    end do
    pset_out%n_beam = count (prt(1:n)%status == PRT_BEAM)
    pset_out%n_in   = count (prt(1:n)%status == PRT_INCOMING)
    pset_out%n_vir  = count (prt(1:n)%status == PRT_RESONANT)
    pset_out%n_out  = count (prt(1:n)%status == PRT_OUTGOING)
    pset_out%n_tot = n
    allocate (pset_out%prt (n))
    do i = 1, n
        call particle_init (pset_out%prt(i), pset%prt(prt(i)%src))
        call particle_reset_status (pset_out%prt(i), prt(i)%status)
        if (prt(i)%orig == 0) then
! This causes nagfor 5.2 (770) Panic
!         call particle_set_parents &
!         (pset_out%prt(i), &
!         map2 (particle_get_parents (pset%prt(prt(i)%src))))
! Workaround
            n_parents = particle_get_n_parents (pset%prt(prt(i)%src))
            parent(1:n_parents) = particle_get_parents (pset%prt(prt(i)%src))
            call particle_set_parents (pset_out%prt(i), &
                                     map2(parent(1:n_parents)))
        else
            call particle_set_parents (pset_out%prt(i), (/ prt(i)%orig /))
        end if
        if (prt(i)%copy == 0) then
! This causes nagfor 5.2 (770) Panic
!         call particle_set_children &
!         (pset_out%prt(i), &
!         map1 (particle_get_children (pset%prt(prt(i)%src))))
! Workaround
            n_children = particle_get_n_children (pset%prt(prt(i)%src))
            child(1:n_children) = particle_get_children (pset%prt(prt(i)%src))
            call particle_set_children (pset_out%prt(i), &
                                     map1(child(1:n_children)))
        else
            call particle_set_children (pset_out%prt(i), (/ prt(i)%copy /))
        end if
    end do
contains
    subroutine append (i)
        integer, intent(in) :: i
        n = n + 1
        if (n > size (prt)) &
            call msg_bug ("Particle set transform to HEPEVT: insufficient space")
        prt(n)%src = i
        prt(n)%status = particle_get_status (pset%prt(i))
        if (map1(i) == 0) then

```

```

        map1(i) = n
    else
        prt(map2(i))%status = PRT_VIRTUAL
        prt(map2(i))%copy = n
        prt(n)%orig = map2(i)
    end if
    map2(i) = n
end subroutine append
end subroutine particle_set_to_hepevt_form

```

Reconstruct the momenta within an interaction object from a particle set. We start from the incoming particles, which should be stored at the beginning. (The number of incoming particles is provided separately, so we do not need to analyze the interaction or particle set for this.) We follow their children down the tree until the interaction is exhausted.

This implies that the particle set may contain a longer tree (e.g., decay chains) than the interaction. The remainder of the tree is simply ignored.

The detailed ordering of the subsequent sub-interactions may differ between the interaction and the particle set. However, within each sub-interaction, the ordering of particles must match.

*(Particles: public)+≡*

```

    public :: particle_set_fill_interaction

```

*(Particles: procedures)+≡*

```

subroutine particle_set_fill_interaction (pset, int, n_in)
    type(particle_set_t), intent(in) :: pset
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: n_in
    logical, dimension(:), allocatable :: p_is_set
    integer, dimension(:), allocatable :: i_parent, j_parent
    integer :: n_tot, k
    n_tot = interaction_get_n_tot (int)
    allocate (p_is_set (n_tot), source = .false.)
    allocate (i_parent (n_in), source = [(k, k = 1, n_in)])
    allocate (j_parent (n_in), source = i_parent)
    do k = 1, n_in
        call fill_subint (i_parent(k), j_parent(k))
    end do
    if (.not. all (p_is_set)) then
        call particle_set_write (pset)
        call interaction_write (int)
        call msg_fatal ("Mismatch between particle set and interaction")
    end if
contains
    recursive subroutine fill_subint (i, j)
        integer, intent(in) :: i, j
        integer, dimension(:), allocatable :: i_child, j_child
        integer :: n_child, n_child_pset, k
        if (p_is_set(i)) return
        call interaction_set_momentum (int, &
            particle_get_momentum (pset%prt(j)), i)
        p_is_set(i) = .true.
        n_child = interaction_get_n_children (int, i)

```

```

n_child_pset = particle_get_n_children (pset%prt(j))
if (n_child /= 0 .and. n_child == n_child_pset) then
  allocate (i_child (n_child))
  allocate (j_child (n_child))
  i_child = interaction_get_children (int, i)
  j_child = particle_get_children (pset%prt(j))
  do k = 1, n_child
    call fill_subint (i_child(k), j_child(k))
  end do
end if
end subroutine fill_subint
end subroutine particle_set_fill_interaction

```

This reconstructs the hard interaction from a particle set. First the particle set is reduced to the incoming partons, then we add the direct children of those. The interaction and particle set should match in their array sizes. If we find a permutation of the particle flavors that matches one of the given flavor states, the assignment of momenta is made accordingly.

```

<CCC Particles: public>+≡
  public :: particle_set_extract_interaction

<CCC Particles: procedures>+≡
  subroutine particle_set_extract_interaction (pset, int, flv_state)
    type(particle_set_t), intent(in) :: pset
    type(interaction_t), intent(inout) :: int
    integer, dimension(:,,:), intent(in) :: flv_state
    integer :: n_in, n_out, n_tot
    integer, dimension(:), allocatable :: status, incoming, outgoing, index
    integer, dimension(:), allocatable :: pdg, perm
    integer :: i
    logical :: ok
    allocate (status (pset%n_tot))
    status = particle_get_status (pset%prt)
    n_in = count (status == PRT_INCOMING)
    allocate (incoming (n_in))
    incoming = pack ((/ (i, i = 1, pset%n_tot) /), status == PRT_INCOMING)
    i = incoming (1)
    n_out = particle_get_n_children (pset%prt(i))
    allocate (outgoing (n_out))
    outgoing = particle_get_children (pset%prt(i))
    n_tot = n_in + n_out
    if (n_in /= interaction_get_n_in (int) &
        .or. n_out /= interaction_get_n_out (int) &
        .or. n_tot /= interaction_get_n_tot (int)) then
      call msg_fatal &
        ("This event does not match the associated process (size)")
      return
    end if
    allocate (index (n_tot), pdg (n_tot), perm (n_tot))
    index(:n_in) = incoming
    index(n_in+1:) = outgoing
    pdg = particle_get_pdg (pset%prt(index))
    call find_flavor_ordering (flv_state, pdg, n_in, perm, ok)
    if (.not. ok) then

```

```

        call particle_set_write (pset)
        call msg_fatal &
            ("This event does not match the associated process (flavors)")
        return
    end if
    do i = 1, n_tot
        call interaction_set_momentum (int, &
            particle_get_momentum (pset%prt(i)), perm(i))
    end do
end subroutine particle_set_extract_interaction

```

Given a particle set with an arbitrary ordering of particles, we need the permutation that yields an ordering that is present in the state matrix. This can be found by examining the flavor table. The result is a permutation of the PDG array that matches a row in the flavor table. If ok is returned false, no match was found.

*(Particles: procedures)*+≡

```

subroutine find_flavor_ordering (flv_state, pdg, n_in, perm, ok)
    integer, dimension(:, :), intent(in) :: flv_state
    integer, dimension(:), intent(in) :: pdg
    integer, intent(in) :: n_in
    integer, dimension(:), intent(out) :: perm
    logical, intent(out) :: ok
    integer :: n_tot, f, i, j, k
    logical, dimension(:), allocatable :: found
    n_tot = size (pdg)
    if (size (flv_state, 1) /= n_tot) then
        ok = .false.
        return
    end if
    do i = 1, n_in
        perm(i) = i
    end do
    allocate (found (n_tot))
    ok = .false.
    do f = 1, size (flv_state, 2)
        call find_ordering_for_this_state (flv_state(:, f))
    end do
contains
    subroutine find_ordering_for_this_state (pdg_state)
        integer, dimension(:), intent(in) :: pdg_state
        found = .false.
        if (all (pdg_state(1:n_in) == pdg(1:n_in))) then
            SCAN_INPUT: do j = n_in + 1, n_tot
                SCAN_STATE: do k = n_in + 1, n_tot
                    if (found(k)) cycle SCAN_STATE
                    if (pdg_state(k) == pdg(j)) then
                        found(k) = .true.
                        perm(j) = k
                        cycle SCAN_INPUT
                    end if
                end do SCAN_STATE
            end do
        end if
    end subroutine find_ordering_for_this_state
end subroutine find_flavor_ordering

```

```

        end do SCAN_INPUT
        ok = .true.
    end if
    end subroutine find_ordering_for_this_state
end subroutine find_flavor_ordering

```

This procedure reconstructs an array of vertex indices from the parent-child information in the particle entries, according to the HepMC scheme. For each particle, we determine which vertex it comes from and which vertex it goes to. We return the two arrays and the maximum vertex index.

For each particle in the list, we first check its parents. If for any parent the vertex where it goes to is already known, this vertex index is assigned as the current 'from' vertex. Otherwise, a new index is created, assigned as the current 'from' vertex, and as the 'to' vertex for all parents.

Then, the analogous procedure is done for the children.

```

(Particles: public) +=
    public :: particle_set_assign_vertices

(Particles: procedures) +=
    subroutine particle_set_assign_vertices &
        (particle_set, v_from, v_to, n_vertices)
        type(particle_set_t), intent(in) :: particle_set
        integer, dimension(:), intent(out) :: v_from, v_to
        integer, intent(out) :: n_vertices
        integer, dimension(:), allocatable :: parent, child
        integer :: n_parents, n_children, vf, vt
        integer :: i, j, v
        v_from = 0
        v_to = 0
        vf = 0
        vt = 0
        do i = 1, particle_set%n_tot
            n_parents = particle_get_n_parents (particle_set%prt(i))
            if (n_parents /= 0) then
                allocate (parent (n_parents))
                parent = particle_get_parents (particle_set%prt(i))
                SCAN_PARENTS: do j = 1, size (parent)
                    v = v_to(parent(j))
                    if (v /= 0) then
                        v_from(i) = v; exit SCAN_PARENTS
                    end if
                end do SCAN_PARENTS
                if (v_from(i) == 0) then
                    vf = vf + 1; v_from(i) = vf
                    v_to(parent) = vf
                end if
                deallocate (parent)
            end if
            n_children = particle_get_n_children (particle_set%prt(i))
            if (n_children /= 0) then
                allocate (child (n_children))
                child = particle_get_children (particle_set%prt(i))
                SCAN_CHILDREN: do j = 1, size (child)

```

```

        v = v_from(child(j))
        if (v /= 0) then
            v_to(i) = v; exit SCAN_CHILDREN
        end if
    end do SCAN_CHILDREN
    if (v_to(i) == 0) then
        vt = vt + 1; v_to(i) = vt
        v_from(child) = vt
    end if
    deallocate (child)
end if
end do
n_vertices = max (vf, vt)
end subroutine particle_set_assign_vertices

```

### 8.2.4 Expression interface

This converts a `particle_set` object as defined here to a more concise `subevt` object that can be used as the event root of an expression. In particular, the latter lacks virtual particles, spin correlations and parent-child relations.

```

(Particles: public)+≡
    public :: particle_set_to_subevt

(Particles: procedures)+≡
    subroutine particle_set_to_subevt (particle_set, subevt)
        type(particle_set_t), intent(in), target :: particle_set
        type(subevt_t), intent(out) :: subevt
        type(particle_t), pointer :: prt
        integer :: i, k
        integer, dimension(2) :: hel
        call subevt_init &
            (subevt, particle_set%n_beam + particle_set%n_in + particle_set%n_out)
        k = 0
        do i = 1, particle_set%n_tot
            prt => particle_set%prt(i)
            select case (particle_get_status (prt))
            case (PRT_BEAM)
                k = k + 1
                call subevt_set_beam (subevt, k, &
                    particle_get_pdg (prt), &
                    particle_get_momentum (prt), &
                    particle_get_p2 (prt))
            case (PRT_INCOMING)
                k = k + 1
                call subevt_set_incoming (subevt, k, &
                    particle_get_pdg (prt), &
                    particle_get_momentum (prt), &
                    particle_get_p2 (prt))
            case (PRT_OUTGOING)
                k = k + 1
                call subevt_set_outgoing (subevt, k, &
                    particle_get_pdg (prt), &
                    particle_get_momentum (prt), &

```

```

        particle_get_p2 (prt))
    end select
    select case (particle_get_status (prt))
    case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
        if (prt%polarization == PRT_DEFINITE_HELICITY) then
            if (helicity_is_diagonal (prt%hel)) then
                hel = helicity_get (prt%hel)
                call subevt_polarize (subevt, k, hel(1))
            end if
        end if
    end select
end do
end subroutine particle_set_to_subevt

```

This replaces the particle\_set

*(Particles: public)*+≡

```
public :: particle_set_replace
```

*(Particles: procedures)*+≡

```

subroutine particle_set_replace (particle_set, newprt)
    type(particle_set_t), intent(inout) :: particle_set
    type(particle_t), intent(in), dimension(:), allocatable :: newprt
    if (allocated (particle_set%prt)) deallocate (particle_set%prt)
    allocate (particle_set%prt(size (newprt)))
    particle_set%prt = newprt
    particle_set%n_tot = size (newprt)
    particle_set%n_beam = count (particle_get_status (newprt) == PRT_BEAM)
    particle_set%n_in = count (particle_get_status (newprt) == PRT_INCOMING)
    particle_set%n_out = count (particle_get_status (newprt) == PRT_OUTGOING)
    particle_set%n_vir = particle_set%n_tot &
        - particle_set%n_beam - particle_set%n_in - particle_set%n_out
end subroutine particle_set_replace

```

## 8.2.5 Auxiliary stuff

Write a separator line.

*(Particles: procedures)*+≡

```

subroutine write_separator (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat (" ", 72)
end subroutine write_separator

subroutine write_separator_double (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("=", 72)
end subroutine write_separator_double

```

Eliminate numerical noise

*(Particles: public)*+≡

```
public :: pacify
```

```

<Particles: interfaces>+≡
  interface pacify
    module procedure pacify_particle
    module procedure pacify_particle_set
  end interface pacify

<Particles: procedures>+≡
  subroutine pacify_particle (prt)
    class(particle_t), intent(inout) :: prt
    real(default) :: e
    e = epsilon (1._default) * energy (prt%p)
    call pacify (prt%p, 10 * e)
    call pacify (prt%p2, 1e4 * e)
  end subroutine pacify_particle

  subroutine pacify_particle_set (pset)
    class(particle_set_t), intent(inout) :: pset
    integer :: i
    do i = 1, pset%n_tot
      call pacify (pset%prt(i))
    end do
  end subroutine pacify_particle_set

```

## 8.2.6 Test

Set up a chain of production and decay and factorize the result into particles.  
The process is  $d\bar{d} \rightarrow Z \rightarrow q\bar{q}$ .

```

<CCC Particles: public>+≡
  public :: particles_test

<CCC Particles: procedures>+≡
  subroutine particles_test
    use os_interface, only: os_data_t
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    type(vector4_t), dimension(3) :: p
    type(interaction_t), target :: int1, int2
    type(quantum_numbers_mask_t) :: qn_mask_conn, qn_rest
    type(evaluator_t), target :: eval
    type(interaction_t), pointer :: int
    type(particle_set_t) :: particle_set1, particle_set2
    type(particle_set_t) :: particle_set3, particle_set4
    type(hepmc_event_t) :: hepmc_event
    type(hepmc_iostream_t) :: iostream
    type(subevt_t) :: subevt
    logical :: ok
    integer :: u, iostat
    print *, "*** Read model file"
  end subroutine particles_test

```



```

call syntax_model_file_init ()
call model_list_read_model &
    (var_str("SM"), var_str("SM.mdl"), os_data, model)
print *
print *, "**** Setup production process ****"
call interaction_init (int1, 2, 0, 1, set_relations=.true.)
call flavor_init (flv, (/1, -1, 23/), model)
call helicity_init (hel(3), 1, 1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default, 0.25_default))
call helicity_init (hel(3),-1, 1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default,-0.25_default))
call helicity_init (hel(3),-1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 0, 0)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.5_default, 0._default))
call interaction_freeze (int1)
p(1) = vector4_moving (45._default, 45._default, 3)
p(2) = vector4_moving (45._default,-45._default, 3)
p(3) = p(1) + p(2)
call interaction_set_momenta (int1, p)
print *
print *, "**** Setup decay process ****"
call interaction_init (int2, 1, 0, 2, set_relations=.true.)
call flavor_init (flv, (/23, 1, -1/), model)
call color_init_col_acl (col, (/ 0, 501, 0 /), (/ 0, 0, 501 /))
call helicity_init (hel, (/ 1, 1, 1/), (/ 1, 1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, (/ 1, 1, 1/), (/ -1,-1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default, 0.1_default))
call helicity_init (hel, (/ -1,-1,-1/), (/ 1, 1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default,-0.1_default))
call helicity_init (hel, (/ -1,-1,-1/), (/ -1,-1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, (/ 0, 1,-1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, (/ 0, 1,-1/), (/ 0,-1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0,-1, 1/))

```

```

call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call flavor_init (flv, (/23, 2, -2/), model)
call helicity_init (hel, (/ 0, 1,-1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0,-1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call interaction_freeze (int2)
p(2) = vector4_moving (45._default, 45._default, 2)
p(3) = vector4_moving (45._default,-45._default, 2)
call interaction_set_momenta (int2, p)
call interaction_set_source_link (int2, 1, int1, 3)
call interaction_write (int1)
call interaction_write (int2)
print *
print *, "*** Concatenate production and decay ***"
call evaluator_init_product (eval, int1, int2, qn_mask_conn, &
    connections_are_resonant=.true.)
call evaluator_receive_momenta (eval)
call evaluator_evaluate (eval)
call evaluator_write (eval)
print *
print *, "*** Factorize as subevent (complete, polarized) ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set1, ok, int, int, FM_FACTOR_HELICITY, &
    (/0.2_default, 0.2_default/), .false., .true.)
call particle_set_write (particle_set1)
print *
print *, "*** Write this to HEPEUP and print as LHEF format ***"
call les_houches_events_write_header ()
call heprup_init ((/2212, 2212/), (/7.e3_default, 7.e3_default/), &
    n_processes=1, unweighted=.true., negative_weights=.false.)
call heprup_set_process_parameters (1, 1)
call heprup_write_lhef ()
call particle_set_fill_hepeup (particle_set1)
call hepeup_write_lhef ()
call les_houches_events_write_footer ()
print *
print *, "*** Factorize as subevent (in/out only, selected helicity) ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set2, ok, int, int, FM_SELECT_HELICITY, &
    (/0.9_default, 0.9_default/), .false., .false.)
call particle_set_write (particle_set2)
call particle_set_final (particle_set2)
print *
print *, "*** Factorize as subevent (complete, selected helicity) ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set2, ok, int, int, FM_SELECT_HELICITY, &
    (/0.7_default, 0.7_default/), .false., .true.)

```

```

call particle_set_write (particle_set2)
print *
print *, "*** Write to HepMC, print, and output to particles_test.hepmc.dat ***"
call hepmc_event_init (hepmc_event, 11, 127)
call particle_set_fill_hepmc_event (particle_set2, hepmc_event)
call hepmc_event_print (hepmc_event)
call hepmc_iostream_open_out &
    (iostream , var_str ("particles_test.hepmc.dat"))
call hepmc_iostream_write_event (iostream, hepmc_event)
call hepmc_iostream_close (iostream)
print *
print *, "*** Recover from HepMC file ***"
call particle_set_final (particle_set2)
call hepmc_event_final (hepmc_event)
call hepmc_event_init (hepmc_event)
call hepmc_iostream_open_in &
    (iostream , var_str ("particles_test.hepmc.dat"))
call hepmc_iostream_read_event (iostream, hepmc_event, ok)
call hepmc_iostream_close (iostream)
call particle_set_init (particle_set2, &
    hepmc_event, model, PRT_DEFINITE_HELICITY)
call particle_set_write (particle_set2)
print *
print *, "*** Factorize (complete, polarized, correlated); write and read again ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set3, ok, int, int, FM_FACTOR_HELICITY, &
    (/0.7_default, 0.7_default/), .true., .true.)
call particle_set_write (particle_set3)
!!! !!! The raw writing seems not to be working.
!!! u = free_unit ()
!!! open (u, action="readwrite", form="unformatted", status="scratch")
!!! call particle_set_write_raw (particle_set3, u)
!!! rewind (u)
!!! call particle_set_read_raw (particle_set4, u, iostat=iostat)
!!! close (u)
!!! print *
!!! call particle_set_write (particle_set4)
!!! print *
!!! print *, "*** Transform to a subvt object ***"
!!! call particle_set_to_subvt (particle_set4, subvt)
!!! call subvt_write (subvt)
print *
print *, "*** Cleanup ***"
call particle_set_final (particle_set1)
call particle_set_final (particle_set2)
call particle_set_final (particle_set3)
call particle_set_final (particle_set4)
call evaluator_final (eval)
call interaction_final (int1)
call interaction_final (int2)
call hepmc_event_final (hepmc_event)
end subroutine particles_test

```

## Chapter 9

# Initial State

### 9.1 Beams for collisions and decays

```
<beams.f90>≡  
  <File header>  
  
  module beams  
  
    <Use kinds>  
    <Use strings>  
    use constants !NODEP!  
    <Use file utils>  
    use diagnostics !NODEP!  
    use lorentz !NODEP!  
    use md5  
    use models  
    use flavors  
    use colors  
    use polarizations  
    use quantum_numbers  
    use state_matrices  
    use interactions  
  
    <Standard module head>  
  
    <Beams: public>  
  
    <Beams: types>  
  
    <Beams: interfaces>  
  
    contains  
  
    <Beams: procedures>  
  
  end module beams
```

### 9.1.1 Beam data

The beam data type contains beam data for one or two beams, depending on whether we are dealing with beam collisions or particle decay. In addition, it holds the c.m. energy `sqrts`, the Lorentz transformation `L` that transforms the c.m. system into the lab system, and the pair of c.m. momenta.

```

(Beams: public)≡
    public :: beam_data_t

(Beams: types)≡
    type :: beam_data_t
        logical :: initialized = .false.
        integer :: n = 0
        type(flavor_t), dimension(:), allocatable :: flv
        real(default), dimension(:), allocatable :: mass
        type(polarization_t), dimension(:), allocatable :: pol
        logical :: lab_is_cm_frame = .true.
        type(vector4_t), dimension(:), allocatable :: p_cm
        type(vector4_t), dimension(:), allocatable :: p
        type(lorentz_transformation_t), pointer :: L_cm_to_lab => null ()
        real(default) :: sqrts = 0
        character(32) :: md5sum = ""
    contains
        (Beams: beam_data: TBP)
    end type beam_data_t

```

Generic initializer. This is called by the specific initializers below. Initialize either for decay or for collision.

```

(Beams: procedures)≡
    subroutine beam_data_init (beam_data, n)
        type(beam_data_t), intent(out) :: beam_data
        integer, intent(in) :: n
        beam_data%n = n
        allocate (beam_data%flv (n))
        allocate (beam_data%mass (n))
        allocate (beam_data%pol (n))
        allocate (beam_data%p_cm (n))
        allocate (beam_data%p (n))
        beam_data%initialized = .true.
    end subroutine beam_data_init

```

Finalizer: needed for the polarization components of the beams.

```

(Beams: public)+≡
    public :: beam_data_final

(Beams: procedures)+≡
    subroutine beam_data_final (beam_data)
        type(beam_data_t), intent(inout) :: beam_data
        beam_data%initialized = .false.
        if (allocated (beam_data%pol)) call polarization_final (beam_data%pol)
        if (associated (beam_data%L_cm_to_lab)) deallocate (beam_data%L_cm_to_lab)
    end subroutine beam_data_final

```

The verbose (default) version is for debugging. The short version is for screen output in the UI.

```

<Beams: public>+≡
    public :: beam_data_write

<Beams: beam data: TBP>≡
    procedure :: write => beam_data_write

<Beams: procedures>+≡
    subroutine beam_data_write (beam_data, unit, verbose, write_md5sum)
        class(beam_data_t), intent(in) :: beam_data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose, write_md5sum
        integer :: prt_name_len
        logical :: verb, write_md5
        integer :: u
        u = output_unit (unit); if (u < 0) return
        verb = .false.; if (present (verbose)) verb = verbose
        write_md5 = verb; if (present (write_md5sum)) write_md5 = write_md5sum
        if (.not. beam_data%initialized) then
            write (u, "(1x,A)") "Beam data: [undefined]"
            return
        end if
        prt_name_len = maxval (len (flavor_get_name (beam_data%flv)))
        select case (beam_data%n)
        case (1)
            write (u, "(1x,A)") "Beam data (decay):"
            if (verb) then
                call write_prt (1)
                call polarization_write (beam_data%pol(1), u)
                write (u, *) "R.f. momentum:"
                call vector4_write (beam_data%p_cm(1), u)
                write (u, *) "Lab momentum:"
                call vector4_write (beam_data%p(1), u)
            else
                call write_prt (1)
            end if
        case (2)
            write (u, "(1x,A)") "Beam data (collision):"
            if (verb) then
                call write_prt (1)
                call polarization_write (beam_data%pol(1), u)
                call write_prt (2)
                call polarization_write (beam_data%pol(2), u)
                call write_sqrts
                write (u, *) "C.m. momenta:"
                call vector4_write (beam_data%p_cm(1), u)
                call vector4_write (beam_data%p_cm(2), u)
                write (u, *) "Lab momenta:"
                call vector4_write (beam_data%p(1), u)
                call vector4_write (beam_data%p(2), u)
            else
                call write_prt (1)
                call write_prt (2)
                call write_sqrts
            end if
        end select
    end subroutine beam_data_write

```

```

        end if
    end select
    if (associated (beam_data%L_cm_to_lab)) &
        call lorentz_transformation_write (beam_data%L_cm_to_lab, u)
    if (write_md5) then
        write (u, *) "MD5 sum: ", beam_data%md5sum
    end if
contains
    subroutine write_sqrts
        character(80) :: sqrts_str
        write (sqrts_str, "(ES19.12)") beam_data%sqrts
        write (u, "(3x,A)") "sqrts = " // trim (adjustl (sqrts_str)) // " GeV"
    end subroutine write_sqrts
    subroutine write_prt (i)
        integer, intent(in) :: i
        character(80) :: name_str, mass_str
        write (name_str, "(A)") char (flavor_get_name (beam_data%flv(i)))
        write (mass_str, "(ES13.7)") beam_data%mass(i)
        write (u, "(3x,A)") name_str(:prt_name_len) // " (mass = " &
            // trim (adjustl (mass_str)) // " GeV)"
    end subroutine write_prt
end subroutine beam_data_write

```

Return initialization status:

```

<Beams: public>+≡
    public :: beam_data_are_valid

<Beams: procedures>+≡
    function beam_data_are_valid (beam_data) result (flag)
        logical :: flag
        type(beam_data_t), intent(in) :: beam_data
        flag = beam_data%initialized
    end function beam_data_are_valid

```

Check whether beam data agree with the current values of relevant parameters.

```

<Beams: public>+≡
    public :: beam_data_check_scattering

<Beams: procedures>+≡
    subroutine beam_data_check_scattering (beam_data, sqrts)
        type(beam_data_t), intent(in) :: beam_data
        real(default), intent(in), optional :: sqrts
        if (beam_data_are_valid (beam_data)) then
            if (present (sqrts)) then
                if (sqrts /= beam_data%sqrts) then
                    call msg_error ("Current setting of sqrts is inconsistent " &
                        // "with beam setup (ignored).")
                end if
            end if
        else
            call msg_bug ("Beam setup: invalid beam data")
        end if
    end subroutine beam_data_check_scattering

```

Return the number of beams (1 for decays, 2 for collisions).

```
<Beams: public>+≡  
    public :: beam_data_get_n_in  
  
<Beams: procedures>+≡  
    function beam_data_get_n_in (beam_data) result (n_in)  
        integer :: n_in  
        type(beam_data_t), intent(in) :: beam_data  
        n_in = beam_data%n  
    end function beam_data_get_n_in
```

Return the beam flavor

```
<Beams: public>+≡  
    public :: beam_data_get_flavor  
  
<Beams: procedures>+≡  
    function beam_data_get_flavor (beam_data) result (flv)  
        type(flavor_t), dimension(:), allocatable :: flv  
        type(beam_data_t), intent(in) :: beam_data  
        allocate (flv (beam_data%n))  
        flv = beam_data%flv  
    end function beam_data_get_flavor
```

Return the beam energies

```
<Beams: public>+≡  
    public :: beam_data_get_energy  
  
<Beams: procedures>+≡  
    function beam_data_get_energy (beam_data) result (e)  
        real(default), dimension(:), allocatable :: e  
        type(beam_data_t), intent(in) :: beam_data  
        allocate (e (beam_data%n))  
        if (beam_data%initialized) then  
            e = energy (beam_data%p)  
        else  
            e = 0  
        end if  
    end function beam_data_get_energy
```

Return the c.m. energy.

```
<Beams: public>+≡  
    public :: beam_data_get_sqrts  
  
<Beams: procedures>+≡  
    function beam_data_get_sqrts (beam_data) result (sqrts)  
        real(default) :: sqrts  
        type(beam_data_t), intent(in) :: beam_data  
        sqrts = beam_data%sqrts  
    end function beam_data_get_sqrts
```

Return a MD5 checksum for beam data. If no checksum is present (because beams have not been initialized), compute the checksum of the sqrts value.

```
<Beams: public>+≡  
    public :: beam_data_get_md5sum
```



```

(Beams: procedures)+≡
function beam_data_get_md5sum (beam_data, sqrts) result (md5sum_beams)
  type(beam_data_t), intent(in) :: beam_data
  real(default), intent(in) :: sqrts
  character(32) :: md5sum_beams
  character(80) :: buffer
  if (beam_data%md5sum /= "") then
    md5sum_beams = beam_data%md5sum
  else
    write (buffer, *) sqrts
    md5sum_beams = md5sum (buffer)
  end if
end function beam_data_get_md5sum

```

### 9.1.2 Initializers: collisions

This is the simplest one: just the two flavors, c.m. energy, polarization. Color is inferred from flavor. Beam momenta and c.m. momenta coincide.

```

(Beams: public)+≡
public :: beam_data_init_sqrts

(Beams: procedures)+≡
subroutine beam_data_init_sqrts (beam_data, &
  sqrts, flv, pol, p_cm, theta, phi)
  type(beam_data_t), intent(out) :: beam_data
  real(default), intent(in) :: sqrts
  type(flavor_t), dimension(:), intent(in) :: flv
  type(polarization_t), dimension(:), intent(in), optional :: pol
  real(default), intent(in), optional :: p_cm, theta, phi
  real(default), dimension(size(flv)) :: E, p
  call beam_data_init (beam_data, size (flv))
  beam_data%sqrts = sqrts
  beam_data%lab_is_cm_frame = &
    .not. present (p_cm) .and. .not. present (theta)
  select case (beam_data%n)
  case (1)
    if (present (p_cm)) then
      E = sqrt (sqrts**2 + p_cm**2)
      p = p_cm
    else
      E = sqrts; p = 0
    end if
    beam_data%p_cm = vector4_moving (E, p, 3)
    beam_data%p = beam_data%p_cm
  case (2)
    beam_data%p_cm = colliding_momenta (sqrts, flavor_get_mass (flv))
    beam_data%p = colliding_momenta (sqrts, flavor_get_mass (flv), p_cm)
  end select
  call beam_data_finish_initialization &
    (beam_data, flv, pol, theta, phi)
end subroutine beam_data_init_sqrts

```

This version uses the momenta of both beams. Note that both momenta are positive numbers. (Applies only to collisions.)

We allow for a crossing angle  $\alpha$ . The crossing angle is applied such that both incoming particle momenta are rotated in the  $zx$  plane by  $\alpha/2$ , from the  $\pm z$  axis towards the  $x$  axis.

```

(Beams: public)+≡
    public :: beam_data_init_momenta

(Beams: procedures)+≡
    subroutine beam_data_init_momenta (beam_data, p, flv, pol, alpha, theta, phi)
        type(beam_data_t), intent(out) :: beam_data
        real(default), dimension(2), intent(in) :: p
        type(flavor_t), dimension(2), intent(in) :: flv
        type(polarization_t), dimension(2), intent(in), optional :: pol
        real(default), intent(in), optional :: alpha, theta, phi
        real(default), dimension(2) :: m, e
        real(default) :: ca, sa
        call beam_data_init (beam_data, 2)
        m = flavor_get_mass (flv)
        e = sqrt (p**2 + m**2)
        beam_data%p(1) = vector4_moving (e(1), p(1), 3)
        beam_data%p(2) = vector4_moving (e(2),-p(2), 3)
        if (present (alpha)) then
            ca = cos (alpha / 2)
            sa = sin (alpha / 2)
            beam_data%p(1) = rotation (ca, sa, 2) * beam_data%p(1)
            beam_data%p(2) = rotation (ca,-sa, 2) * beam_data%p(2)
            beam_data%sqrts = &
                sqrt (2 * (e(1)*e(2) + p(1)*p(2)*cos(alpha)) + m(1)**2 + m(2)**2)
        else
            beam_data%sqrts = &
                sqrt (2 * (e(1)*e(2) + p(1)*p(2)) + m(1)**2 + m(2)**2)
        end if
        beam_data%p_cm = colliding_momenta (beam_data%sqrts, m)
        call beam_data_finish_initialization (beam_data, flv, pol, theta, phi)
    end subroutine beam_data_init_momenta

```

Final steps: If requested, rotate the beams in the lab frame, and set the beam-data components.

```

(Beams: procedures)+≡
    subroutine beam_data_finish_initialization &
        (beam_data, flv, pol, theta, phi)
        type(beam_data_t), intent(inout) :: beam_data
        type(flavor_t), dimension(:), intent(in) :: flv
        type(polarization_t), dimension(:), intent(in), optional :: pol
        real(default), intent(in), optional :: theta, phi
        integer :: i
        if (present (theta)) then
            beam_data%p = rotation (theta, 2) * beam_data%p
            if (present (phi)) then
                beam_data%p = rotation (phi, 3) * beam_data%p
            end if
        end if
    end subroutine beam_data_finish_initialization

```

```

do i = 1, beam_data%n
  beam_data%flv(i) = flv(i)
  beam_data%mass(i) = flavor_get_mass (flv(i))
  if (present (pol)) then
    beam_data%pol(i) = pol(i)
  else
    call polarization_init_unpolarized (beam_data%pol(i), flv(i))
  end if
end do
call beam_data_compute_md5sum (beam_data)
if (logging) call beam_data_write (beam_data)
end subroutine beam_data_finish_initialization

```

The MD5 sum is stored within the beam-data record, so it can be checked for integrity in subsequent runs.

*(Beams: procedures)*+≡

```

subroutine beam_data_compute_md5sum (beam_data)
  type(beam_data_t), intent(inout) :: beam_data
  integer :: unit
  unit = free_unit ()
  open (unit = unit, status = "scratch", action = "readwrite")
  call beam_data_write (beam_data, unit, write_md5sum = .false., &
    verbose = .true.)
  rewind (unit)
  beam_data%md5sum = md5sum (unit)
  close (unit)
end subroutine beam_data_compute_md5sum

```

### 9.1.3 Initializers: decays

This is the simplest one: decay in rest frame. We need just flavor and polarization. Color is inferred from flavor. Beam momentum and c.m. momentum coincide.

*(Beams: public)*+≡

```

public :: beam_data_init_decay

```

*(Beams: procedures)*+≡

```

subroutine beam_data_init_decay &
  (beam_data, flv, pol, p_cm, theta, phi)
  type(beam_data_t), intent(out) :: beam_data
  type(flavor_t), dimension(1), intent(in) :: flv
  type(polarization_t), dimension(1), intent(in), optional :: pol
  real(default), intent(in), optional :: p_cm, theta, phi
  real(default), dimension(1) :: m
  type(polarization_t), dimension(1) :: polarization
  m = flavor_get_mass (flv)
  if (present (pol)) then
    call beam_data_init_sqrts &
      (beam_data, m(1), flv, pol, p_cm, theta, phi)
  else
    call polarization_init_trivial (polarization(1), flv(1))
    call beam_data_init_sqrts &

```

```

        (beam_data, m(1), flv, polarization, p_cm, theta, phi)
    end if
end subroutine beam_data_init_decay

```

#### 9.1.4 Data access

Post-initialization changes to the polarization:

```

(Beams: public)+≡
    public :: beam_data_set_polarization
    public :: beam_data_kill_polarization

(Beams: procedures)+≡
    subroutine beam_data_set_polarization (beam_data, pol)
        type(beam_data_t), intent(inout) :: beam_data
        type(polarization_t), dimension(:), intent(in) :: pol
        integer :: i
        if (size (pol) /= beam_data%n) call msg_bug ( &
            "beam_data_set_polarization_pol: initial state multiplicity mismatch")
        !do i = 1, beam_data%n
        !    call polarization_final (beam_data%pol(i))
        !end do
        beam_data%pol = pol
        call beam_data_compute_md5sum (beam_data)
    end subroutine beam_data_set_polarization

    subroutine beam_data_kill_polarization (beam_data)
        type(beam_data_t), intent(inout) :: beam_data
        if (beam_data%n == 1) then
            call polarization_init_trivial (beam_data%pol(1), beam_data%flv(1))
        else
            call polarization_init_unpolarized (beam_data%pol(1), beam_data%flv(1))
            call polarization_init_unpolarized (beam_data%pol(2), beam_data%flv(2))
        end if
    end subroutine beam_data_kill_polarization

```

#### 9.1.5 Sanity check

After the beams have been set, the initial-particle masses may have been modified. This can be checked here.

```

(Beams: public)+≡
    public :: beam_data_masses_are_consistent

(Beams: procedures)+≡
    function beam_data_masses_are_consistent (beam_data) result (flag)
        logical :: flag
        type(beam_data_t), intent(in) :: beam_data
        flag = all (beam_data%mass == flavor_get_mass (beam_data%flv))
    end function beam_data_masses_are_consistent

```

### 9.1.6 The beams type

Beam objects are interaction objects that contain the actual beam data including polarization and density matrix. For collisions, the beam object actually contains two beams.

```

(Beams: public)+≡
    public :: beam_t

(Beams: types)+≡
    type :: beam_t
    private
        type(interaction_t) :: int
    end type beam_t

```

The constructor contains code that converts beam data into the (entangled) particle-pair quantum state. First, we set the number of particles and polarization mask. (The polarization mask is handed over to all later interactions, so if helicity is diagonal or absent, this fact is used when constructing the hard-interaction events.) Then, we construct the entangled state that combines helicity, flavor and color of the two particles (where flavor and color are unique, while several helicity states are possible). Then, we transfer this state together with the associated values from the spin density matrix into the `interaction_t` object.

```

(Beams: public)+≡
    public :: beam_init

(Beams: procedures)+≡
    subroutine beam_init (beam, beam_data)
        type(beam_t), intent(out) :: beam
        type(beam_data_t), intent(in), target :: beam_data
        type(quantum_numbers_mask_t), dimension(beam_data%n) :: mask
        type(state_matrix_t), target :: state_hel, state_fc, state_tmp
        type(state_iterator_t) :: it_hel, it_tmp
        type(quantum_numbers_t), dimension(:), allocatable :: qn
        mask = new_quantum_numbers_mask (.false., .false., &
            .not. polarization_is_polarized (beam_data%pol), &
            mask_hd = polarization_is_diagonal (beam_data%pol))
        call interaction_init &
            (beam%int, 0, 0, beam_data%n, mask=mask, store_values=.true.)
        call combine_polarization_states (beam_data%pol, state_hel)
        allocate (qn (beam_data%n))
        call quantum_numbers_init &
            (qn, beam_data%flv, color_from_flavor (beam_data%flv, 1))
        call state_matrix_init (state_fc)
        call state_matrix_add_state (state_fc, qn)
        call merge_state_matrices (state_hel, state_fc, state_tmp)
        call state_iterator_init (it_hel, state_hel)
        call state_iterator_init (it_tmp, state_tmp)
        do while (state_iterator_is_valid (it_hel))
            call interaction_add_state (beam%int, &
                state_iterator_get_quantum_numbers (it_tmp), &
                value=state_iterator_get_matrix_element (it_hel))
            call state_iterator_advance (it_hel)
            call state_iterator_advance (it_tmp)
        end do
    end subroutine beam_init

```

```

end do
call interaction_freeze (beam%int)
call interaction_set_momenta &
    (beam%int, beam_data%p, outgoing = .true.)
call state_matrix_final (state_hel)
call state_matrix_final (state_fc)
call state_matrix_final (state_tmp)
end subroutine beam_init

```

Finalizer:

```

(Beams: public)+≡
    public :: beam_final

(Beams: procedures)+≡
    elemental subroutine beam_final (beam)
        type(beam_t), intent(inout) :: beam
        call interaction_final (beam%int)
    end subroutine beam_final

```

I/O:

```

(Beams: public)+≡
    public :: beam_write

(Beams: procedures)+≡
    subroutine beam_write (beam, unit, verbose, show_momentum_sum, show_mass)
        type(beam_t), intent(in) :: beam
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
        integer :: u
        u = output_unit (unit); if (u < 0) return
        select case (interaction_get_n_out (beam%int))
        case (1); write (u, *) "Decaying particle:"
        case (2); write (u, *) "Colliding beams:"
        end select
        call interaction_write &
            (beam%int, unit, verbose, show_momentum_sum, show_mass)
    end subroutine beam_write

```

Defined assignment: deep copy

```

(Beams: public)+≡
    public :: assignment(=)

(Beams: interfaces)≡
    interface assignment(=)
        module procedure beam_assign
    end interface

(Beams: procedures)+≡
    subroutine beam_assign (beam_out, beam_in)
        type(beam_t), intent(out) :: beam_out
        type(beam_t), intent(in) :: beam_in
        beam_out%int = beam_in%int
    end subroutine beam_assign

```

### 9.1.7 Inherited procedures

```
<Beams: public>+≡
  public :: interaction_set_source_link

<Beams: interfaces>+≡
  interface interaction_set_source_link
    module procedure interaction_set_source_link_beam
  end interface

<Beams: procedures>+≡
  subroutine interaction_set_source_link_beam (int, i, beam1, i1)
    type(interaction_t), intent(inout) :: int
    type(beam_t), intent(in), target :: beam1
    integer, intent(in) :: i, i1
    call interaction_set_source_link (int, i, beam1%int, i1)
  end subroutine interaction_set_source_link_beam
```

### 9.1.8 Accessing contents

Return the interaction component – as a pointer, to avoid any copying.

```
<Beams: public>+≡
  public :: beam_get_int_ptr

<Beams: procedures>+≡
  function beam_get_int_ptr (beam) result (int)
    type(interaction_t), pointer :: int
    type(beam_t), intent(in), target :: beam
    int => beam%int
  end function beam_get_int_ptr
```

Set beam momenta directly. (Used for cascade decays.)

```
<Beams: public>+≡
  public :: beam_set_momenta

<Beams: procedures>+≡
  subroutine beam_set_momenta (beam, p)
    type(beam_t), intent(inout) :: beam
    type(vector4_t), dimension(:), intent(in) :: p
    call interaction_set_momenta (beam%int, p)
  end subroutine beam_set_momenta
```

### 9.1.9 Test

```
<Beams: public>+≡
  public :: beam_test

<Beams: procedures>+≡
  subroutine beam_test ()
    use os_interface, only: os_data_t
    type(os_data_t) :: os_data
    type(beam_data_t), target :: beam_data
    type(beam_t) :: beam
```

```

real(default) :: sqrts
type(flavor_t), dimension(2) :: flv
type(polarization_t), dimension(2) :: pol
type(model_t), pointer :: model
print *, "*** Read model file"
call syntax_model_file_init ()
call model_list_read_model &
      (var_str("SM"), var_str("SM.mdl"), os_data, model)
call syntax_model_file_final ()
print *
print *, "*** Scattering"
sqrts = 500
call flavor_init (flv, ((/1,-1/)), model)
call polarization_init_circular (pol(1), flv(1), 0.5_default)
call polarization_init_transversal (pol(2), flv(2), 0._default, 1._default)
call beam_data_init_sqrts (beam_data, sqrts, flv, pol)
call beam_data_write (beam_data)
print *
call beam_init (beam, beam_data)
call beam_write (beam)
call beam_final (beam)
call beam_data_final (beam_data)
print *
print *, "*** Decay"
call flavor_init (flv(1), 23, model)
call polarization_init_longitudinal (pol(1), flv(1), 0.4_default)
call beam_data_init_decay (beam_data, flv(1:1), pol(1:1))
call beam_data_write (beam_data)
print *
call beam_init (beam, beam_data)
call beam_write (beam)
call beam_final (beam)
call beam_data_final (beam_data)
end subroutine beam_test

```



## Chapter 10

# Spectra and structure functions

### 10.1 Tools

This module contains auxiliary procedures that can be accessed by the structure function code.

```
(sf_aux.f90)≡  
  ⟨File header⟩  
  
  module sf_aux  
  
    ⟨Use kinds⟩  
    use constants, only: twopi !NODEP!  
    ⟨Use file utils⟩  
    use unit_tests  
  
    use lorentz !NODEP!  
  
    ⟨Standard module head⟩  
  
    ⟨SF aux: public⟩  
  
    ⟨SF aux: parameters⟩  
  
    ⟨SF aux: types⟩  
  
    contains  
  
    ⟨SF aux: procedures⟩  
  
    ⟨SF aux: tests⟩  
  
  end module sf_aux
```

### 10.1.1 Momentum splitting

Let us consider first an incoming parton with momentum  $k$  and invariant mass squared  $s = k^2$  that splits into two partons with momenta  $q, p$  and invariant masses  $t = q^2$  and  $u = p^2$ . (This is an abuse of the Mandelstam notation.  $t$  is actually the momentum transfer, assuming that  $p$  is radiated and  $q$  initiates the hard process.) The energy is split among the partons such that if  $E = k^0$ , we have  $q^0 = xE$  and  $p^0 = \bar{x}E$ , where  $\bar{x} \equiv 1 - x$ .

We define the angle  $\theta$  as the polar angle of  $p$  w.r.t. the momentum axis of the incoming momentum  $k$ . Ignoring azimuthal angle, we can write the four-momenta in the basis  $(E, p_T, p_L)$  as

$$k = \begin{pmatrix} E \\ 0 \\ p \end{pmatrix}, \quad p = \begin{pmatrix} \bar{x}E \\ \bar{x}\bar{p} \sin \theta \\ \bar{x}\bar{p} \cos \theta \end{pmatrix}, \quad q = \begin{pmatrix} xE \\ -\bar{x}\bar{p} \sin \theta \\ p - \bar{x}\bar{p} \cos \theta \end{pmatrix}, \quad (10.1)$$

where the first two mass-shell conditions are

$$p^2 = E^2 - s, \quad \bar{p}^2 = E^2 - \frac{u}{\bar{x}^2}. \quad (10.2)$$

The second condition implies that, for positive  $u$ ,  $\bar{x}^2 > u/E^2$ , or equivalently

$$x < 1 - \sqrt{u}/E. \quad (10.3)$$

We are interested in the third mass-shell conditions:  $s$  and  $u$  are fixed, so we need  $t$  as a function of  $\cos \theta$ :

$$t = -2\bar{x}(E^2 - p\bar{p} \cos \theta) + s + u. \quad (10.4)$$

Solving for  $\cos \theta$ , we get

$$\cos \theta = \frac{2\bar{x}E^2 + t - s - u}{2\bar{x}p\bar{p}}. \quad (10.5)$$

We can compute  $\sin^2 \theta$  numerically as  $\sin^2 \theta = 1 - \cos^2 \theta$ , but it is important to reexpress this in view of numerical stability. To this end, we first determine the bounds for  $t$ . The cosine must be between  $-1$  and  $1$ , so the bounds are

$$t_0 = -2\bar{x}(E^2 + p\bar{p}) + s + u, \quad (10.6)$$

$$t_1 = -2\bar{x}(E^2 - p\bar{p}) + s + u. \quad (10.7)$$

Computing  $\sin^2 \theta$  from  $\cos \theta$  above, we observe that the numerator is a quadratic polynomial in  $t$  which has the zeros  $t_0$  and  $t_1$ , while the common denominator is given by  $(2\bar{x}p\bar{p})^2$ . Hence, we can write

$$\sin^2 \theta = -\frac{(t - t_0)(t - t_1)}{(2\bar{x}p\bar{p})^2} \quad \text{and} \quad \cos \theta = \frac{(t - t_0) + (t - t_1)}{4\bar{x}p\bar{p}}, \quad (10.8)$$

which is free of large cancellations near  $t = t_0$  or  $t = t_1$ .

If all is massless, i.e.,  $s = u = 0$ , this simplifies to

$$t_0 = -4\bar{x}E^2, \quad t_1 = 0, \quad (10.9)$$

$$\sin^2 \theta = -\frac{t}{\bar{x}E^2} \left( 1 + \frac{t}{4\bar{x}E^2} \right), \quad \cos \theta = 1 + \frac{t}{2\bar{x}E^2}. \quad (10.10)$$

Here is the implementation. First, we define a container for the kinematical integration limits and some further data.

```

<SF aux: public>≡
  public :: splitting_data_t

<SF aux: types>≡
  type :: splitting_data_t
    private
    logical :: collinear = .false.
    real(default) :: x0 = 0
    real(default) :: x1
    real(default) :: t0
    real(default) :: t1
    real(default) :: phi0 = 0
    real(default) :: phi1 = twopi
    real(default) :: E, p, s, u, m2
    real(default) :: x, xb, pb
    real(default) :: t = 0
    real(default) :: phi = 0
    contains
    <SF aux: splitting data: TBP>
  end type splitting_data_t

```

I/O for debugging:

```

<SF aux: splitting data: TBP>≡
  procedure :: write => splitting_data_write

<SF aux: procedures>≡
  subroutine splitting_data_write (d, unit)
    class(splitting_data_t), intent(in) :: d
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, "(A)") "Splitting data:"
    write (u, "(2x,A,L1)") "collinear = ", d%collinear
1   format (2x,A,1x,ES15.8)
    write (u, 1) "x0  =", d%x0
    write (u, 1) "x   =", d%x
    write (u, 1) "xb  =", d%xb
    write (u, 1) "x1  =", d%x1
    write (u, 1) "t0  =", d%t0
    write (u, 1) "t   =", d%t
    write (u, 1) "t1  =", d%t1
    write (u, 1) "phi0 =", d%phi0
    write (u, 1) "phi  =", d%phi
    write (u, 1) "phi1 =", d%phi1
    write (u, 1) "E    =", d%E
    write (u, 1) "p    =", d%p
    write (u, 1) "pb   =", d%pb
    write (u, 1) "s    =", d%s
    write (u, 1) "u    =", d%u
    write (u, 1) "m2   =", d%m2
  end subroutine splitting_data_write

```

### 10.1.2 Constant data

This is the initializer for the data. The input consists of the incoming momentum, its invariant mass squared, and the invariant mass squared of the radiated particle.  $m_2$  is the *physical* mass squared of the outgoing particle. The  $t$  bounds depend on the chosen  $x$  value and cannot be determined yet.

```

(SF aux: splitting data: TBP)+≡
  procedure :: init => splitting_data_init

(SF aux: procedures)+≡
  subroutine splitting_data_init (d, k, mk2, mr2, mo2, collinear)
    class(splitting_data_t), intent(out) :: d
    type(vector4_t), intent(in) :: k
    real(default), intent(in) :: mk2, mr2, mo2
    logical, intent(in), optional :: collinear
    if (present (collinear)) d%collinear = collinear
    d%E = energy (k)
    d%x1 = 1 - sqrt (max (mr2, 0._default)) / d%E
    d%p = sqrt (d%E**2 - mk2)
    d%s = mk2
    d%u = mr2
    d%m2 = mo2
  end subroutine splitting_data_init

```

Retrieve the  $x$  bounds, if needed for  $x$  sampling. Generating an  $x$  value is done by the caller, since this is the part that depends on the nature of the structure function.

```

(SF aux: splitting data: TBP)+≡
  procedure :: get_x_bounds => splitting_get_x_bounds

(SF aux: procedures)+≡
  function splitting_get_x_bounds (d) result (x)
    class(splitting_data_t), intent(in) :: d
    real(default), dimension(2) :: x
    x = (/ d%x0, d%x1 /)
  end function splitting_get_x_bounds

```

Now set the momentum fraction and compute  $t_0$  and  $t_1$ .

[The calculation of  $t_1$  is subject to numerical problems. The exact formula is ( $s = m_i^2$ ,  $u = m_r^2$ )

$$t_1 = -2\bar{x}E^2 + m_i^2 + m_r^2 + 2\bar{x}\sqrt{E^2 - m_i^2}\sqrt{E^2 - m_r^2/\bar{x}^2}. \quad (10.11)$$

The structure-function paradigm is useful only if  $E \gg m_i, m_r$ . In a Taylor expansion for large  $E$ , the leading term cancels. The expansion of the square roots (to subleading order) yields

$$t_1 = xm_i^2 - \frac{x}{\bar{x}}m_r^2. \quad (10.12)$$

There are two cases of interest:  $m_i = m_o$  and  $m_r = 0$ ,

$$t_1 = xm_o^2 \quad (10.13)$$

and  $m_i = m_r$  and  $m_o = 0$ ,

$$t_1 = -\frac{x^2}{\bar{x}} m_i^2. \quad (10.14)$$

In both cases,  $t_1 \leq m_o^2$ .]

That said, it turns out that taking the  $t_1$  evaluation at face value leads to less problems than the approximation. We express the angles in terms of  $t - t_0$  and  $t - t_1$ . Numerical noise in  $t_1$  can then be tolerated.

```

<SF aux: splitting data: TBP>+≡
  procedure :: set_t_bounds => splitting_set_t_bounds

<SF aux: procedures>+≡
  elemental subroutine splitting_set_t_bounds (d, x, xb)
    class(splitting_data_t), intent(inout) :: d
    real(default), intent(in), optional :: x, xb
    real(default) :: tp, tm
    if (present (x)) d%x = x
    if (present (xb)) d%xb = xb
    if (d%xb /= 0) then
      d%pb = sqrt (max (d%E**2 - d%u / d%xb**2, 0._default))
    else
      d%pb = 0
    end if
    tp = -2 * d%xb * d%E**2 + d%s + d%u
    tm = -2 * d%xb * d%p * d%pb
    d%t0 = tp + tm
    d%t1 = tp - tm
    !!! JRR: WK please check
    !!! where does this old version come from? Supposed to
    !!! be numerically more stable earlier, I guess, but
    !!! is the formula correct!?
    !   if (d%xb /= 0) then
    !     d%t1 = d%x * (d%s - d%u / d%xb)
    !   else
    !     d%t1 = 0
    !   end if
    d%t = d%t1
  end subroutine splitting_set_t_bounds

```

### 10.1.3 Sampling recoil

Compute a value for the momentum transfer  $t$ , using a random number  $r$ . We assume a logarithmic distribution for  $t - m^2$ , corresponding to the propagator  $1/(t - m^2)$  with the physical mass  $m$  for the outgoing particle. Optionally, we can narrow the kinematical bounds.

If all three masses in the splitting vanish, the upper limit for  $t$  is zero. In that case, the  $t$  value is set to zero and the splitting will be collinear.

```

<SF aux: splitting data: TBP>+≡
  procedure :: sample_t => splitting_sample_t

<SF aux: procedures>+≡
  subroutine splitting_sample_t (d, r, t0, t1)
    class(splitting_data_t), intent(inout) :: d

```

```

real(default), intent(in) :: r
real(default), intent(in), optional :: t0, t1
real(default) :: tt0, tt1, tt0m, tt1m
if (d%collinear) then
  d%t = d%t1
else
  tt0 = d%t0; if (present (t0)) tt0 = max (t0, tt0)
  tt1 = d%t1; if (present (t1)) tt1 = min (t1, tt1)
  tt0m = tt0 - d%m2
  tt1m = tt1 - d%m2
  if (tt0m < 0 .and. tt1m < 0 .and. abs(tt0m) > &
      epsilon(tt0m) .and. abs(tt1m) > epsilon(tt0m)) then
    d%t = d%m2 + tt0m * exp (r * log (tt1m / tt0m))
  else
    d%t = tt1
  end if
end if
end subroutine splitting_sample_t

```

The inverse operation: Given  $t$ , we recover the value of  $r$  that would have produced this value.

```

<SF aux: splitting data: TBP>+≡
  procedure :: inverse_t => splitting_inverse_t

<SF aux: procedures>+≡
  subroutine splitting_inverse_t (d, r, t0, t1)
    class(splitting_data_t), intent(in) :: d
    real(default), intent(out) :: r
    real(default), intent(in), optional :: t0, t1
    real(default) :: tt0, tt1, tt0m, tt1m
    if (d%collinear) then
      r = 0
    else
      tt0 = d%t0; if (present (t0)) tt0 = max (t0, tt0)
      tt1 = d%t1; if (present (t1)) tt1 = min (t1, tt1)
      tt0m = tt0 - d%m2
      tt1m = tt1 - d%m2
      if (tt0m < 0 .and. tt1m < 0) then
        r = log ((d%t - d%m2) / tt0m) / log (tt1m / tt0m)
      else
        r = 0
      end if
    end if
  end subroutine splitting_inverse_t

```

This is trivial, but provided for convenience:

```

<SF aux: splitting data: TBP>+≡
  procedure :: sample_phi => splitting_sample_phi

<SF aux: procedures>+≡
  subroutine splitting_sample_phi (d, r)
    class(splitting_data_t), intent(inout) :: d
    real(default), intent(in) :: r
    if (d%collinear) then

```

```

        d%phi = 0
    else
        d%phi = (1-r) * d%phi0 + r * d%phi1
    end if
end subroutine splitting_sample_phi

```

Inverse:

```

<SF aux: splitting data: TBP>+≡
    procedure :: inverse_phi => splitting_inverse_phi

<SF aux: procedures>+≡
    subroutine splitting_inverse_phi (d, r)
        class(splitting_data_t), intent(in) :: d
        real(default), intent(out) :: r
        if (d%collinear) then
            r = 0
        else
            r = (d%phi - d%phi0) / (d%phi1 - d%phi0)
        end if
    end subroutine splitting_inverse_phi

```

#### 10.1.4 Splitting

In this function, we actually perform the splitting. The incoming momentum  $k$  is split into (if no recoil)  $q_1 = (1 - x)k$  and  $q_2 = xk$ .

Apart from the splitting data, we need the incoming momentum  $k$ , the momentum transfer  $t$ , and the azimuthal angle  $\phi$ . The momentum fraction  $x$  is already known here.

Alternatively, we can split without recoil. The azimuthal angle is irrelevant, and the momentum transfer is always equal to the upper limit  $t_1$ , so the polar angle is zero. Obviously, if there are nonzero masses it is not possible to keep both energy-momentum conservation and at the same time all particles on shell. We choose for dropping the on-shell condition here.

```

<SF aux: splitting data: TBP>+≡
    procedure :: split_momentum => splitting_split_momentum

<SF aux: procedures>+≡
    function splitting_split_momentum (d, k) result (q)
        class(splitting_data_t), intent(in) :: d
        type(vector4_t), dimension(2) :: q
        type(vector4_t), intent(in) :: k
        real(default) :: st2, ct2, st, ct, cp, sp
        type(lorentz_transformation_t) :: rot
        real(default) :: tt0, tt1, den
        type(vector3_t) :: kk, q1, q2
        if (d%collinear) then
            if (d%s == 0 .and. d%u == 0) then
                q(1) = d%xb * k
                q(2) = d%x * k
            else
                kk = space_part (k)
                q1 = d%xb * (d%pb / d%p) * kk

```

```

        q2 = kk - q1
        q(1) = vector4_moving (d%xb * d%E, q1)
        q(2) = vector4_moving (d%x * d%E, q2)
    end if
else
    den = 2 * d%xb * d%p * d%pb
    tt0 = max (d%t - d%t0, 0._default)
    tt1 = min (d%t - d%t1, 0._default)
    if (den**2 <= epsilon(den)) then
        st2 = 1
    else
        st2 = - (tt0 * tt1) / den ** 2
    end if
    if (st2 > 1) then
        st2 = 1
    end if
    ct2 = 1 - st2
    st = sqrt (max (st2, 0._default))
    ct = sqrt (max (ct2, 0._default))
    sp = sin (d%phi)
    cp = cos (d%phi)
    rot = rotation_to_2nd (3, space_part (k))
    q1 = vector3_moving (d%xb * d%pb * (/ st * cp, st * sp, ct /))
    q2 = vector3_moving (d%p, 3) - q1
    q(1) = rot * vector4_moving (d%xb * d%E, q1)
    q(2) = rot * vector4_moving (d%x * d%E, q2)
end if
end function splitting_split_momentum

```

Momenta generated by splitting will in general be off-shell. They are on-shell only if they are collinear and massless. This subroutine puts them on shell by brute force, violating either momentum or energy conservation. The direction of three-momentum is always retained.

If the energy is below mass shell, we return a zero momentum.

```

<SF aux: parameters>≡
    integer, parameter, public :: KEEP_ENERGY = 0, KEEP_MOMENTUM = 1

<SF aux: public>+≡
    public :: on_shell

<SF aux: procedures>+≡
    elemental subroutine on_shell (p, m2, keep)
        type(vector4_t), intent(inout) :: p
        real(default), intent(in) :: m2
        integer, intent(in) :: keep
        real(default) :: E, E2, pn
        select case (keep)
        case (KEEP_ENERGY)
            E = energy (p)
            E2 = E ** 2
            if (E2 >= m2) then
                pn = sqrt (E2 - m2)
                p = vector4_moving (E, pn * direction (space_part (p)))
            else

```



```

        p = vector4_null
    end if
    case (KEEP_MOMENTUM)
        E = sqrt (space_part (p) ** 2 + m2)
        p = vector4_moving (E, space_part (p))
    end select
end subroutine on_shell

```

### 10.1.5 Recovering the splitting

This is the inverse problem. We have on-shell momenta and want to deduce the splitting parameters  $x$ ,  $t$ , and  $\phi$ .

```

<SF aux: splitting data: TBP>+≡
    procedure :: recover => splitting_recover

<SF aux: procedures>+≡
    subroutine splitting_recover (d, k, q, keep)
        class(splitting_data_t), intent(inout) :: d
        type(vector4_t), intent(in) :: k, q
        integer, intent(in) :: keep
        type(lorentz_transformation_t) :: rot
        type(vector4_t) :: q0, k0
        real(default) :: p1, p2, p3, pt2, pp2, pl
        real(default) :: aux, den, norm
        real(default) :: st2, ct2, ct
        rot = inverse (rotation_to_2nd (3, space_part (k)))
        q0 = rot * q
        p1 = vector4_get_component (q0, 1)
        p2 = vector4_get_component (q0, 2)
        p3 = vector4_get_component (q0, 3)
        pt2 = p1 ** 2 + p2 ** 2
        pp2 = p1 ** 2 + p2 ** 2 + p3 ** 2
        pl = abs (p3)
        k0 = vector4_moving (d%E, d%p, 3)
        select case (keep)
        case (KEEP_ENERGY)
            d%x = energy (q0) / d%E
            d%xb = 1 - d%x
            call d%set_t_bounds ()
            if (.not. d%collinear) then
                aux = (d%xb * d%pb) ** 2 * pp2 - d%p ** 2 * pt2
                den = d%p ** 2 - (d%xb * d%pb) ** 2
                if (aux >= 0 .and. den > 0) then
                    norm = (d%p * pl + sqrt (aux)) / den
                else
                    norm = 1
                end if
            end if
        case (KEEP_MOMENTUM)
            d%xb = sqrt (space_part (k0 - q0) ** 2 + d%u) / d%E
            d%x = 1 - d%xb
            call d%set_t_bounds ()
            norm = 1

```

```

end select
if (d%collinear) then
  d%t = d%t1
  d%phi = 0
else
  if ((d%xb * d%pb * norm)**2 < epsilon(d%xb)) then
    st2 = 1
  else
    st2 = pt2 / (d%xb * d%pb * norm ) ** 2
  end if
  if (st2 > 1) then
    st2 = 1
  end if
  ct2 = 1 - st2
  ct = sqrt (max (ct2, 0._default))
  if (ct /= -1) then
    d%t = d%t1 - 2 * d%xb * d%p * d%pb * st2 / (1 + ct)
  else
    d%t = d%t0
  end if
  if (p1 /= 0 .or. p2 /= 0) then
    d%phi = atan2 (-p2, -p1)
  else
    d%phi = 0
  end if
end if
end subroutine splitting_recover

```

### 10.1.6 Extract data

```

<SF aux: splitting data: TBP>+≡
  procedure :: get_x => splitting_get_x

<SF aux: procedures>+≡
  function splitting_get_x (sd) result (x)
    class(splitting_data_t), intent(in) :: sd
    real(default) :: x
    x = sd%x
  end function splitting_get_x

```

### 10.1.7 Unit tests

```

<SF aux: public>+≡
  public :: sf_aux_test

<SF aux: tests>≡
  subroutine sf_aux_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <SF aux: execute tests>
  end subroutine sf_aux_test

```

## Momentum splitting: massless radiation

Compute momentum splitting for generic kinematics. It turns out that for  $x = 0.5$ , where  $t - m^2$  is the geometric mean between its upper and lower bounds (this can be directly seen from the logarithmic distribution in the function `sample_t` for  $r \equiv x = 1 - x = 0.5$ ), we arrive at an exact number  $t = -0.15$  for the given input values.

```
(SF aux: execute tests)≡
  call test (sf_aux_1, "sf_aux_1", &
    "massless radiation", &
    u, results)

(SF aux: tests)+≡
  subroutine sf_aux_1 (u)
    integer, intent(in) :: u
    type(splitting_data_t) :: sd
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q, q0
    real(default) :: E, mk, mp, mq
    real(default) :: x, r1, r2, r1o, r2o
    real(default) :: k2, q0_2, q1_2, q2_2

    write (u, "(A)")  "* Test output: sf_aux_1"
    write (u, "(A)")  "*   Purpose: compute momentum splitting"
    write (u, "(A)")  "                               (massless radiated particle)"
    write (u, "(A)")

    E = 1
    mk = 0.3_default
    mp = 0
    mq = mk

    k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
    k2 = k ** 2;  call pacify (k2, 1e-10_default)

    x = 0.6_default
    r1 = 0.5_default
    r2 = 0.125_default

    write (u, "(A)")  "* (1) Non-collinear setup"
    write (u, "(A)")

    call sd%init (k, mk**2, mp**2, mq**2)
    call sd%set_t_bounds (x, 1 - x)
    call sd%sample_t (r1)
    call sd%sample_phi (r2)

    call sd%write (u)

    q = sd%split_momentum (k)
    q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
    q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

    write (u, "(A)")
```

```

write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: s"
write (u, "(2(1x,F11.8))") sd%s, k2

write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") sd%t, q2_2

write (u, "(A)") "Compare: u"
write (u, "(2(1x,F11.8))") sd%u, q1_2

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") sd%x, energy (q(2)) / energy (k)

write (u, "(A)") "Compare: 1-x"
write (u, "(2(1x,F11.8))") sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q0_2 = q0(2) ** 2; call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q0_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"

```

```

write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q0_2 = q0(2) ** 2; call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q0_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

```

```

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2; call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2; call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: s"
write (u, "(2(1x,F11.8))") sd%s, k2

write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") sd%t, q2_2

write (u, "(A)") "Compare: u"
write (u, "(2(1x,F11.8))") sd%u, q1_2

```

```

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") sd%x, energy (q(2)) / energy (k)

write (u, "(A)") "Compare: 1-x"
write (u, "(2(1x,F11.8))") sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q0_2 = q0(2) ** 2; call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q0_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="

```

```

call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q0_2 = q0(2) ** 2; call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q0_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Test output end: sf_aux_1"

end subroutine sf_aux_1

```

### Momentum splitting: massless parton

Compute momentum splitting for generic kinematics. It turns out that for  $x = 0.5$ , where  $t - m^2$  is the geometric mean between its upper and lower bounds, we arrive at an exact number  $t = -0.36$  for the given input values.

```

<SF aux: execute tests>+≡
call test (sf_aux_2, "sf_aux_2", &
  "massless parton", &
  u, results)

<SF aux: tests>+≡
subroutine sf_aux_2 (u)
integer, intent(in) :: u
type(splitting_data_t) :: sd
type(vector4_t) :: k
type(vector4_t), dimension(2) :: q, q0
real(default) :: E, mk, mp, mq

```



```

real(default) :: x, r1, r2, r1o, r2o
real(default) :: k2, q02_2, q1_2, q2_2

write (u, "(A)")  "* Test output: sf_aux_2"
write (u, "(A)")  "*   Purpose: compute momentum splitting"
write (u, "(A)")  "               (massless outgoing particle)"
write (u, "(A)")

E = 1
mk = 0.3_default
mp = mk
mq = 0

k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
k2 = k ** 2;  call pacify (k2, 1e-10_default)

x = 0.6_default
r1 = 0.5_default
r2 = 0.125_default

write (u, "(A)")  "* (1) Non-collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

```

```

write (u, "(A)") "Compare: u"
write (u, "(2(1x,F11.8))") sd%u, q1_2

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") sd%x, energy (q(2)) / energy (k)

write (u, "(A)") "Compare: 1-x"
write (u, "(2(1x,F11.8))") sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

```

```

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")

```

```

call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")

```

```

write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

```

```

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_aux_2"

end subroutine sf_aux_2

```

### Momentum splitting: all massless

Compute momentum splitting for massless kinematics. In the non-collinear case, we need a lower cutoff for  $|t|$ , otherwise a logarithmic distribution is not possible.

```

<SF aux: execute tests>+≡
  call test (sf_aux_3, "sf_aux_3", &
    "massless parton", &
    u, results)

<SF aux: tests>+≡
  subroutine sf_aux_3 (u)
    integer, intent(in) :: u
    type(splitting_data_t) :: sd
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q, q0
    real(default) :: E, mk, mp, mq, qmin, qmax
    real(default) :: x, r1, r2, r1o, r2o
    real(default) :: k2, q02_2, q1_2, q2_2

    write (u, "(A)")  "* Test output: sf_aux_3"
    write (u, "(A)")  "*   Purpose: compute momentum splitting"
    write (u, "(A)")  "               (all massless, q cuts)"
    write (u, "(A)")

    E = 1
    mk = 0
    mp = 0
    mq = 0
    qmin = 1e-2_default
    qmax = 1e0_default

```

```

k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
k2 = k ** 2; call pacify (k2, 1e-10_default)

x = 0.6_default
r1 = 0.5_default
r2 = 0.125_default

write (u, "(A)")  "* (1) Non-collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1, t1 = - qmin ** 2, t0 = - qmax **2)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2; call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2; call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q

```

```

call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")

```



```

write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2
write (u, "(A)")

write (u, "(A)") "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") x, sd%x
write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") q2_2, sd%t

call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)") "Compare: r1"
write (u, "(2(1x,F11.8))") r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)") "Compare: r2"
write (u, "(2(1x,F11.8))") r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)") "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2; call pacify (q1_2, 1e-10_default)

```

```

q2_2 = q(2) ** 2; call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: s"
write (u, "(2(1x,F11.8))") sd%s, k2

write (u, "(A)") "Compare: t"
write (u, "(2(1x,F11.8))") sd%t, q2_2

write (u, "(A)") "Compare: u"
write (u, "(2(1x,F11.8))") sd%u, q1_2

write (u, "(A)") "Compare: x"
write (u, "(2(1x,F11.8))") sd%x, energy (q(2)) / energy (k)

write (u, "(A)") "Compare: 1-x"
write (u, "(2(1x,F11.8))") sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)") "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)") "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)") "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)") "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)") "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))") sd%m2, q02_2

```

```

write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2; call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

```

```

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_aux_3"

end subroutine sf_aux_3

```

## 10.2 Mappings for structure functions

In this module, we provide a wrapper for useful mappings of the unit square that we can apply to a pair of structure functions.

In some cases it is useful, or even mandatory, to map the MC input parameters nontrivially onto a pair of structure functions for the two beams. In all cases considered here, instead of  $x_1$  and  $x_2$  as parameters for the beams, we generate one parameter for the product  $x_1 x_2$  (so it directly corresponds to  $\sqrt{s}$ ) and one parameter related to the quotient.

```

⟨sf_mappings.f90⟩≡
  ⟨File header⟩

  module sf_mappings

    ⟨Use kinds⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use unit_tests

    ⟨Standard module head⟩

    ⟨SF mappings: public⟩

    ⟨SF mappings: parameters⟩

    ⟨SF mappings: types⟩

    ⟨SF mappings: interfaces⟩

    contains

    ⟨SF mappings: procedures⟩

    ⟨SF mappings: tests⟩

  end module sf_mappings

```

### 10.2.1 Base type

First, we define an abstract base type for the mapping. In all cases we need to store the indices of the parameters on which the mapping applies. Additional

parameters can be stored in the extensions of this type.

```

<SF mappings: public>≡
  public :: sf_mapping_t

<SF mappings: types>≡
  type, abstract :: sf_mapping_t
    integer, dimension(:), allocatable :: i
    contains
    <SF mappings: sf mapping: TBP>
  end type sf_mapping_t

```

The output routine is deferred:

```

<SF mappings: sf mapping: TBP>≡
  procedure (sf_mapping_write), deferred :: write

<SF mappings: interfaces>≡
  abstract interface
    subroutine sf_mapping_write (object, unit)
      import
      class(sf_mapping_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine sf_mapping_write
  end interface

```

Initializer for the base type. The array of parameter indices is allocated but initialized to zero.

```

<SF mappings: sf mapping: TBP>+≡
  procedure :: base_init => sf_mapping_base_init

<SF mappings: procedures>≡
  subroutine sf_mapping_base_init (mapping, n_par)
    class(sf_mapping_t), intent(out) :: mapping
    integer, intent(in) :: n_par
    allocate (mapping%i (n_par))
    mapping%i = 0
  end subroutine sf_mapping_base_init

```

Computation: the values **p** are the input parameters, the values **r** are the output parameters. The Jacobian is **f**. We modify only the two parameters indicated by the indices **i**.

```

<SF mappings: sf mapping: TBP>+≡
  procedure (sf_mapping_compute), deferred :: compute

<SF mappings: interfaces>+≡
  abstract interface
    subroutine sf_mapping_compute (mapping, r, f, p)
      import
      class(sf_mapping_t), intent(in) :: mapping
      real(default), dimension(:), intent(out) :: r
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p
    end subroutine sf_mapping_compute
  end interface

```

The inverse mapping. Use `r` to reconstruct `p` and also compute `f`.

```

<SF mappings: sf mapping: TBP>+≡
  procedure (sf_mapping_inverse), deferred :: inverse
<SF mappings: interfaces>+≡
  abstract interface
    subroutine sf_mapping_inverse (mapping, r, f, p)
      import
      class(sf_mapping_t), intent(in) :: mapping
      real(default), dimension(:), intent(in) :: r
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p
    end subroutine sf_mapping_inverse
  end interface

```

### 10.2.2 Integral

This is a consistency check for the self-tests: the integral over the unit square should be unity. We estimate this by a simple binning and adding up the values; this should be sufficient for a self-test.

The argument is the requested number of sampling points. We take the square root for binning in both dimensions, so the precise number might be different.

```

<SF mappings: sf mapping: TBP>+≡
  procedure :: integral => sf_mapping_integral
<SF mappings: procedures>+≡
  function sf_mapping_integral (mapping, n_calls) result (integral)
    class(sf_mapping_t), intent(in) :: mapping
    integer, intent(in) :: n_calls
    real(default) :: integral
    integer :: n_bin, i, j
    real(default), dimension(2) :: p, r
    real(default) :: dx, f, s
    n_bin = nint (sqrt (real (n_calls)))
    dx = 1._default / n_bin
    s = 0
    do i = 1, n_bin
      p(1) = i * dx - dx/2
      do j = 1, n_bin
        p(2) = j * dx - dx/2
        call mapping%compute (r, f, p)
        s = s + f
      end do
    end do
    integral = s / n_bin / n_bin
  end function sf_mapping_integral

```

### 10.2.3 Implementation: standard mapping

This maps the unit square  $(r_1, r_2)$  such that  $p_1$  is the product  $r_1 r_2$ , while  $p_2$  is related to the ratio.

```

<SF mappings: public>+≡
  public :: sf_s_mapping_t

<SF mappings: types>+≡
  type, extends (sf_mapping_t) :: sf_s_mapping_t
    logical :: power_set = .false.
    real(default) :: power = 1._default
  contains
    <SF mappings: sf standard mapping: TBP>
  end type sf_s_mapping_t

```

Output.

```

<SF mappings: sf standard mapping: TBP>≡
  procedure :: write => sf_s_mapping_write

<SF mappings: procedures>+≡
  subroutine sf_s_mapping_write (object, unit)
    class(sf_s_mapping_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)", advance="no") "map"
    if (any (object%i /= 0)) then
      write (u, "('(',IO,',',',IO,')')", advance="no") object%i
    end if
    write (u, "(A,F7.5,A)" ": standard (" , object%power, ")")
  end subroutine sf_s_mapping_write

```

Initialize: index pair and power parameter.

```

<SF mappings: sf standard mapping: TBP>+≡
  procedure :: init => sf_s_mapping_init

<SF mappings: procedures>+≡
  subroutine sf_s_mapping_init (mapping, power)
    class(sf_s_mapping_t), intent(out) :: mapping
    real(default), intent(in), optional :: power
    call mapping%base_init (2)
    if (present (power)) then
      mapping%power_set = .true.
      mapping%power = power
    end if
  end subroutine sf_s_mapping_init

```

Apply mapping.

```

<SF mappings: sf standard mapping: TBP>+≡
  procedure :: compute => sf_s_mapping_compute

<SF mappings: procedures>+≡
  subroutine sf_s_mapping_compute (mapping, r, f, p)
    class(sf_s_mapping_t), intent(in) :: mapping
    real(default), dimension(:), intent(out) :: r
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: p
    real(default), dimension(2) :: r2

```

```

integer :: j
if (mapping%power_set) then
  call map_unit_square (r2, f, p(mapping%i), mapping%power)
else
  call map_unit_square (r2, f, p(mapping%i))
end if
do j = 1, 2
  r(mapping%i(j)) = r2(j)
end do
end subroutine sf_s_mapping_compute

```

Apply inverse.

$\langle SF \text{ mappings: sf standard mapping: TBP} \rangle + \equiv$

```

procedure :: inverse => sf_s_mapping_inverse

```

$\langle SF \text{ mappings: procedures} \rangle + \equiv$

```

subroutine sf_s_mapping_inverse (mapping, r, f, p)
  class(sf_s_mapping_t), intent(in) :: mapping
  real(default), dimension(:), intent(in) :: r
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p
  real(default), dimension(2) :: p2
  integer :: j
  if (mapping%power_set) then
    call map_unit_square_inverse (r(mapping%i), f, p2, mapping%power)
  else
    call map_unit_square_inverse (r(mapping%i), f, p2)
  end if
  do j = 1, 2
    p(mapping%i(j)) = p2(j)
  end do
end subroutine sf_s_mapping_inverse

```

#### 10.2.4 Basic formulas

This mapping of the unit square is appropriate in particular for structure functions which are concentrated at the lower end. Instead of a rectangular grid, one set of grid lines corresponds to constant parton c.m. energy. The other set is chosen such that the jacobian is only mildly singular ( $\ln x$  which is zero at  $x = 1$ ), corresponding to an initial concentration of sampling points at the maximum energy. If **power** is greater than one (the default), points are also concentrated at the lower end.

The formula is (**power**= $\alpha$ ):

$$r_1 = (p_1^{p_2})^\alpha \quad (10.15)$$

$$r_2 = (p_1^{1-p_2})^\alpha \quad (10.16)$$

$$f = \alpha^2 p_1^{\alpha-1} |\log p_1| \quad (10.17)$$



and for the default case  $\alpha = 1$ :

$$r_1 = p_1^{p_2} \quad (10.18)$$

$$r_2 = p_1^{1-p_2} \quad (10.19)$$

$$f = |\log p_1| \quad (10.20)$$

```

<SF mappings: procedures>+≡
subroutine map_unit_square (r, factor, p, power)
  real(default), dimension(2), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), dimension(2), intent(in) :: p
  real(default), intent(in), optional :: power
  real(default) :: xx, yy
  factor = 1
  xx = p(1)
  yy = p(2)
  if (present(power)) then
    if (p(1) > 0 .and. power > 1) then
      xx = p(1)**power
      factor = factor * power * xx / p(1)
    end if
  end if
  if (xx /= 0) then
    r(1) = xx ** yy
    r(2) = xx / r(1)
    factor = factor * abs (log (xx))
  else
    r = 0
  end if
end subroutine map_unit_square

```

This is the inverse mapping.

```

<SF mappings: procedures>+≡
subroutine map_unit_square_inverse (r, factor, p, power)
  real(kind=default), dimension(2), intent(in) :: r
  real(kind=default), intent(out) :: factor
  real(kind=default), dimension(2), intent(out) :: p
  real(kind=default), intent(in), optional :: power
  real(kind=default) :: lg, xx, yy
  factor = 1
  xx = r(1) * r(2)
  if (xx /= 0) then
    lg = log (xx)
    yy = log (r(1)) / lg
    p(2) = yy
    factor = factor * abs (lg)
    if (present(power)) then
      p(1) = xx**(1._default/power)
      factor = factor * power * xx / p(1)
    else
      p(1) = xx
    end if
  else
    r = 0
  end if
end subroutine map_unit_square_inverse

```

```

        p = 0
    end if
end subroutine map_unit_square_inverse

```

### Structure-function channels

A structure-function chain parameterization (channel) may contain a mapping that applies to multiple structure functions. This is described by an extension of the `sf_mapping_t` type. In addition, it may contain mappings that apply to (other) individual structure functions. The details of these mappings are implementation-specific.

The `sf_channel_t` type combines this information. It contains an array of map codes, one for each structure-function entry. The code values are:

**none** MC input parameters  $r$  directly become energy fractions  $x$

**single** default mapping for a single structure-function entry

**multi/s** map  $r \rightarrow x$  such that one MC input parameter is  $\hat{s}/s$

**multi/resonance** as before, adapted to s-channel resonance (not implemented yet)

**multi/on-shell** as before, adapted to an on-shell particle in the s channel (not implemented yet)

```

⟨SF mappings: parameters⟩≡
    integer, parameter :: SFMAP_NONE = 0
    integer, parameter :: SFMAP_SINGLE = 1
    integer, parameter :: SFMAP_MULTI_S = 2
    integer, parameter :: SFMAP_MULTI_RES = 3
    integer, parameter :: SFMAP_MULTI_ONS = 4

```

Then, it contains an allocatable entry for the multi mapping. This entry holds the MC-parameter indices on which the mapping applies (there may be more than one MC parameter per structure-function entry) and any parameters associated with the mapping.

There can be only one multi-mapping per channel.

```

⟨SF mappings: public⟩+≡
    public :: sf_channel_t

⟨SF mappings: types⟩+≡
    type :: sf_channel_t
        integer, dimension(:), allocatable :: map_code
        class(sf_mapping_t), allocatable :: multi_mapping
    contains
        ⟨SF mappings: sf channel: TBP⟩
    end type sf_channel_t

```

The output format prints a single character for each structure-function entry and, if applicable, an account of the mapping parameters.

```

<SF mappings: sf channel: TBP>≡
  procedure :: write => sf_channel_write

<SF mappings: procedures>+≡
  subroutine sf_channel_write (object, unit)
    class(sf_channel_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    if (allocated (object%map_code)) then
      do i = 1, size (object%map_code)
        select case (object%map_code (i))
          case (SFMAP_NONE)
            write (u, "(1x,A)", advance="no") "-"
          case (SFMAP_SINGLE)
            write (u, "(1x,A)", advance="no") "+"
          case (SFMAP_MULTI_S)
            write (u, "(1x,A)", advance="no") "s"
          case (SFMAP_MULTI_RES)
            write (u, "(1x,A)", advance="no") "r"
          case (SFMAP_MULTI_ONS)
            write (u, "(1x,A)", advance="no") "o"
        end select
      end do
    else
      write (u, "(1x,A)", advance="no") "-"
    end if
    if (allocated (object%multi_mapping)) then
      write (u, "(1x,'/')" , advance="no")
      call object%multi_mapping%write (u)
    else
      write (u, *)
    end if
  end subroutine sf_channel_write

```

Initializer for a single `sf_channel` object.

```

<SF mappings: sf channel: TBP>+≡
  procedure :: init => sf_channel_init

<SF mappings: procedures>+≡
  subroutine sf_channel_init (channel, n_strfun)
    class(sf_channel_t), intent(out) :: channel
    integer, intent(in) :: n_strfun
    allocate (channel%map_code (n_strfun))
    channel%map_code = SFMAP_NONE
  end subroutine sf_channel_init

```

Assignment. This merely copies intrinsic assignment, but apparently the latter is bugged in gfortran 4.6.3, causing memory corruption.

```

<SF mappings: sf channel: TBP>+≡
  generic :: assignment (=) => sf_channel_assign
  procedure :: sf_channel_assign

```

```

<SF mappings: procedures>+≡
subroutine sf_channel_assign (copy, original)
  class(sf_channel_t), intent(out) :: copy
  type(sf_channel_t), intent(in) :: original
  allocate (copy%map_code (size (original%map_code)))
  copy%map_code = original%map_code
  if (allocated (original%multi_mapping)) then
    allocate (copy%multi_mapping, source = original%multi_mapping)
  end if
end subroutine sf_channel_assign

```

This initializer allocates an array of channels with common number of structure-function entries, therefore it is not a type-bound procedure.

```

<SF mappings: public>+≡
public :: allocate_sf_channels

<SF mappings: procedures>+≡
subroutine allocate_sf_channels (channel, n_channel, n_strfun)
  type(sf_channel_t), dimension(:), intent(out), allocatable :: channel
  integer, intent(in) :: n_channel
  integer, intent(in) :: n_strfun
  integer :: c
  allocate (channel (n_channel))
  do c = 1, n_channel
    call channel(c)%init (n_strfun)
  end do
end subroutine allocate_sf_channels

```

This marks a given subset of indices as single-mapping.

```

<SF mappings: sf channel: TBP>+≡
procedure :: activate_mapping => sf_channel_activate_mapping

<SF mappings: procedures>+≡
subroutine sf_channel_activate_mapping (channel, i_sf)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  channel%map_code(i_sf) = SFMAP_SINGLE
end subroutine sf_channel_activate_mapping

```

This sets an s-channel multichannel mapping. The length of the `i_sf` array must be 2. The parameter indices are not yet set.

```

<SF mappings: sf channel: TBP>+≡
procedure :: set_s_mapping => sf_channel_set_s_mapping

<SF mappings: procedures>+≡
subroutine sf_channel_set_s_mapping (channel, i_sf, power)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  real(default), intent(in), optional :: power
  channel%map_code(i_sf) = SFMAP_MULTI_S
  allocate (sf_s_mapping_t :: channel%multi_mapping)
  select type (mapping => channel%multi_mapping)
  type is (sf_s_mapping_t)

```

```

        call mapping%init (power)
    end select
end subroutine sf_channel_set_s_mapping

```

Return true if the mapping code at position `i_sf` is `SFMAP_SINGLE`.

```

<SF mappings: sf_channel: TBP>+≡
    procedure :: is_single_mapping => sf_channel_is_single_mapping

<SF mappings: procedures>+≡
    function sf_channel_is_single_mapping (channel, i_sf) result (flag)
        class(sf_channel_t), intent(in) :: channel
        integer, intent(in) :: i_sf
        logical :: flag
        flag = channel%map_code(i_sf) == SFMAP_SINGLE
    end function sf_channel_is_single_mapping

```

Return true if the mapping code at position `i_sf` is any of the `SFMAP_MULTI` mappings.

```

<SF mappings: sf_channel: TBP>+≡
    procedure :: is_multi_mapping => sf_channel_is_multi_mapping

<SF mappings: procedures>+≡
    function sf_channel_is_multi_mapping (channel, i_sf) result (flag)
        class(sf_channel_t), intent(in) :: channel
        integer, intent(in) :: i_sf
        logical :: flag
        select case (channel%map_code(i_sf))
        case (SFMAP_NONE, SFMAP_SINGLE)
            flag = .false.
        case default
            flag = .true.
        end select
    end function sf_channel_is_multi_mapping

```

Return true if there is any nontrivial mapping in any of the channels.

Note: we provide an explicit public function. gfortran 4.6.3 has problems with the alternative implementation as a type-bound procedure for an array base object.

```

<SF mappings: public>+≡
    public :: any_sf_channel_has_mapping

<SF mappings: procedures>+≡
    function any_sf_channel_has_mapping (channel) result (flag)
        type(sf_channel_t), dimension(:), intent(in) :: channel
        logical :: flag
        integer :: c
        flag = .false.
        do c = 1, size (channel)
            flag = flag .or. any (channel(c)%map_code /= SFMAP_NONE)
        end do
    end function any_sf_channel_has_mapping

```

Set a parameter index for an active multi mapping. We assume that the index array is allocated properly.

```

<SF mappings: sf channel: TBP>+≡
  procedure :: set_par_index => sf_channel_set_par_index

<SF mappings: procedures>+≡
  subroutine sf_channel_set_par_index (channel, j, i_par)
    class(sf_channel_t), intent(inout) :: channel
    integer, intent(in) :: j
    integer, intent(in) :: i_par
    channel%multi_mapping%i(j) = i_par
  end subroutine sf_channel_set_par_index

```

### 10.2.5 Unit tests

```

<SF mappings: public>+≡
  public :: sf_mappings_test

<SF mappings: tests>≡
  subroutine sf_mappings_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <SF mappings: execute tests>
  end subroutine sf_mappings_test

```

#### Check standard mapping

Probe the standard mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

```

<SF mappings: execute tests>≡
  call test (sf_mappings_1, "sf_mappings_1", &
    "standard pair mapping", &
    u, results)

<SF mappings: tests>+≡
  subroutine sf_mappings_1 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p, r
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_mappings_1"
    write (u, "(A)")  "* Purpose: probe standard mapping"
    write (u, "(A)")

    allocate (sf_s_mapping_t :: mapping)
    select type (mapping)
    type is (sf_s_mapping_t)
      call mapping%init ()
      mapping%i(1) = 1
      mapping%i(2) = 2

```

```

end select

call mapping%write (u)

write (u, *)
write (u, "(A)") "Probe at (0,0):"
p = [0._default, 0._default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
call mapping%inverse (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
call mapping%inverse (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
call mapping%inverse (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Compute integral:"

```

```

write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)
allocate (sf_s_mapping_t :: mapping)
select type (mapping)
type is (sf_s_mapping_t)
    call mapping%init (power=2._default)
    mapping%i(1) = 1
    mapping%i(2) = 2
end select

write (u, *)
call mapping%write (u)

write (u, *)
write (u, "(A)") "Probe at (0,0):"
p = [0._default, 0._default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
call mapping%inverse (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
call mapping%inverse (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)

```



```

write (u, "(A)") "Probe at (0.1,0.1):"
p = [0.1_default, 0.1_default]

call mapping%compute (r, f, p)
write (u, "(3x,A,2(1x,F7.5))") "p =", p
write (u, "(3x,A,2(1x,F7.5))") "r =", r
write (u, "(3x,A,2(1x,F7.5))") "f =", f

write (u, *)
write (u, "(A)") "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

write (u, "(A)")
write (u, "(A)") "* Test output end: sf_mappings_1"

end subroutine sf_mappings_1

```

## Channel entries

Construct channel entries and print them.

```

<SF mappings: execute tests>+≡
call test (sf_mappings_2, "sf_mappings_2", &
"structure-function mapping channels", &
u, results)

<SF mappings: tests>+≡
subroutine sf_mappings_2 (u)
integer, intent(in) :: u
type(sf_channel_t), dimension(:), allocatable :: channel
integer :: c

write (u, "(A)") "* Test output: sf_mappings_2"
write (u, "(A)") "* Purpose: construct and display &
&mapping-channel objects"
write (u, "(A)")

call allocate_sf_channels (channel, n_channel = 4, n_strfun = 2)
call channel(2)%activate_mapping ([1])
call channel(3)%set_s_mapping ([1,2])
call channel(4)%set_s_mapping ([1,2], power=2._default)

call channel(3)%set_par_index (1, 1)
call channel(3)%set_par_index (2, 4)

call channel(4)%set_par_index (1, 1)
call channel(4)%set_par_index (2, 4)

do c = 1, size (channel)
write (u, "(I0,':')", advance="no") c
call channel(c)%write (u)
end do

write (u, "(A)")

```

```

        write (u, "(A)")  "* Test output end: sf_mappings_2"

    end subroutine sf_mappings_2

```

## 10.3 Structure function base

```

⟨sf_base.f90⟩≡
  ⟨File header⟩

  module sf_base

    ⟨Use kinds⟩
    ⟨Use strings⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use models
    use flavors
    use helicities
    use quantum_numbers
    use state_matrices
    use interactions
    use evaluators
    use particles
    use beams
    use sf_aux
    use sf_mappings

    ⟨Standard module head⟩

    ⟨SF base: public⟩

    ⟨SF base: parameters⟩

    ⟨SF base: types⟩

    ⟨SF base: interfaces⟩

    ⟨SF base: test types⟩

    contains

    ⟨SF base: procedures⟩

    ⟨SF base: tests⟩

  end module sf_base

```

### 10.3.1 Abstract structure-function data type

This type should hold all configuration data for a specific type of structure function. The base object is empty; the implementations will fill it.

```
<SF base: public>≡
    public :: sf_data_t

<SF base: types>≡
    type, abstract :: sf_data_t
    contains
        <SF base: sf data: TBP>
    end type sf_data_t
```

Output.

```
<SF base: sf data: TBP>≡
    procedure (sf_data_write), deferred :: write

<SF base: interfaces>≡
    abstract interface
        subroutine sf_data_write (data, unit, verbose)
            import
            class(sf_data_t), intent(in) :: data
            integer, intent(in), optional :: unit
            logical, intent(in), optional :: verbose
        end subroutine sf_data_write
    end interface
```

Return the number of input parameters that determine the structure function.

```
<SF base: sf data: TBP>+≡
    procedure (sf_data_get_int), deferred :: get_n_par

<SF base: interfaces>+≡
    abstract interface
        function sf_data_get_int (data) result (n)
            import
            class(sf_data_t), intent(in) :: data
            integer :: n
        end function sf_data_get_int
    end interface
```

Return the outgoing particle PDG codes for the current setup.

```
<SF base: sf data: TBP>+≡
    procedure (sf_data_get_pdg_out), deferred :: get_pdg_out

<SF base: interfaces>+≡
    abstract interface
        function sf_data_get_pdg_out (data) result (pdg_out)
            import
            class(sf_data_t), intent(in) :: data
            integer, dimension(:), allocatable :: pdg_out
        end function sf_data_get_pdg_out
    end interface
```

Allocate a matching structure function interaction object and properly initialize it.

Note: We should insist on `intent(out)` for the second argument, but gfortran 4.6.3 does not like it. In any case, the interaction should enter this deallocated.

```

<SF base: sf data: TBP>+≡
  procedure (sf_data_allocate_sf_int), deferred :: allocate_sf_int
<SF base: interfaces>+≡
  abstract interface
    subroutine sf_data_allocate_sf_int (data, sf_int)
      import
      class(sf_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
    end subroutine sf_data_allocate_sf_int
  end interface

```

### 10.3.2 Structure-function chain configuration

This is the data type that the `processes` module uses for setting up its structure-function chain. For each structure function described by the beam data, there is an entry. The `i` array indicates the beam(s) to which this structure function applies, and the `data` object contains the actual configuration data.

```

<SF base: public>+≡
  public :: sf_config_t
<SF base: types>+≡
  type :: sf_config_t
    integer, dimension(:), allocatable :: i
    class(sf_data_t), allocatable :: data
  contains
    <SF base: sf config: TBP>
  end type sf_config_t

```

Output:

```

<SF base: sf config: TBP>≡
  procedure :: write => sf_config_write
<SF base: procedures>≡
  subroutine sf_config_write (object, unit)
    class(sf_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (allocated (object%i)) then
      write (u, "(1x,A,2(1x,I0))") "Structure-function configuration: &
        &beam(s)", object%i
      if (allocated (object%data)) call object%data%write (u)
    else
      write (u, "(1x,A)") "Structure-function configuration: [undefined]"
    end if
  end subroutine sf_config_write

```

Initialize.

```

<SF base: sf config: TBP>+=
  procedure :: init => sf_config_init

<SF base: procedures>+=
  subroutine sf_config_init (sf_config, i_beam, sf_data)
    class(sf_config_t), intent(out) :: sf_config
    integer, dimension(:), intent(in) :: i_beam
    class(sf_data_t), intent(in) :: sf_data
    allocate (sf_config%i (size (i_beam)), source = i_beam)
    allocate (sf_config%data, source = sf_data)
  end subroutine sf_config_init

```

### 10.3.3 Structure-function instance

The `sf_int_t` data type contains an `interaction_t` object (it is an extension of this type) and a pointer to the `sf_data_t` configuration data. This interaction, or copies of it, is used to implement structure-function kinematics and dynamics in the context of process evaluation.

The status code `status` tells whether the interaction is undefined, has defined kinematics (but matrix elements invalid), or is completely defined. There is also a status code for failure. The implementation is responsible for updating the status.

The entries `mi2`, `mr2`, and `mo2` hold the squared invariant masses of the incoming, radiated, and outgoing particle, respectively. They are supposed to be set upon initialization, but could also be varied event by event.

If the radiated or outgoing mass is nonzero, we may need to apply an on-shell projection. The projection mode is stored as `on_shell_mode`.

The array `beam_index` is the list of beams on which this structure function applies (1, 2, or both). The arrays `incoming`, `radiated`, and `outgoing` contain the indices of the respective particle sets within the interaction, for convenient lookup. The array `par_index` indicates the MC input parameters that this entry will use up in the structure-function chain.

In the abstract base type, we do not implement the data pointer. This allows us to restrict its type in the implementations.

```

<SF base: public>+=
  public :: sf_int_t

<SF base: types>+=
  type, abstract, extends (interaction_t) :: sf_int_t
    integer :: status = SF_UNDEFINED
    real(default), dimension(:), allocatable :: mi2
    real(default), dimension(:), allocatable :: mr2
    real(default), dimension(:), allocatable :: mo2
    integer :: on_shell_mode = KEEP_ENERGY
    logical :: qmin_defined = .false.
    logical :: qmax_defined = .false.
    real(default), dimension(:), allocatable :: qmin
    real(default), dimension(:), allocatable :: qmax
    integer, dimension(:), allocatable :: beam_index
    integer, dimension(:), allocatable :: incoming

```

```

integer, dimension(:), allocatable :: radiated
integer, dimension(:), allocatable :: outgoing
integer, dimension(:), allocatable :: par_index
contains
<SF base: sf_int: TBP>
end type sf_int_t

```

Status codes. The codes that refer to links, masks, and connections, apply to structure-function chains only.

The status codes are public.

```

<SF base: parameters>≡
integer, parameter, public :: SF_UNDEFINED = 0
integer, parameter, public :: SF_INITIAL = 1
integer, parameter, public :: SF_DONE_LINKS = 2
integer, parameter, public :: SF_FAILED_MASK = 3
integer, parameter, public :: SF_DONE_MASK = 4
integer, parameter, public :: SF_FAILED_CONNECTIONS = 5
integer, parameter, public :: SF_DONE_CONNECTIONS = 6
integer, parameter, public :: SF_SEED_KINEMATICS = 10
integer, parameter, public :: SF_FAILED_KINEMATICS = 11
integer, parameter, public :: SF_DONE_KINEMATICS = 12
integer, parameter, public :: SF_FAILED_EVALUATION = 13
integer, parameter, public :: SF_EVALUATED = 20

```

Write a string version of the status code:

```

<SF base: procedures>+≡
subroutine write_sf_status (status, u)
integer, intent(in) :: status
integer, intent(in) :: u
select case (status)
case (SF_UNDEFINED)
write (u, "(1x,'[',A,']')") "undefined"
case (SF_INITIAL)
write (u, "(1x,'[',A,']')") "initialized"
case (SF_DONE_LINKS)
write (u, "(1x,'[',A,']')") "links set"
case (SF_FAILED_MASK)
write (u, "(1x,'[',A,']')") "mask mismatch"
case (SF_DONE_MASK)
write (u, "(1x,'[',A,']')") "mask set"
case (SF_FAILED_CONNECTIONS)
write (u, "(1x,'[',A,']')") "connections failed"
case (SF_DONE_CONNECTIONS)
write (u, "(1x,'[',A,']')") "connections set"
case (SF_SEED_KINEMATICS)
write (u, "(1x,'[',A,']')") "incoming momenta set"
case (SF_FAILED_KINEMATICS)
write (u, "(1x,'[',A,']')") "kinematics failed"
case (SF_DONE_KINEMATICS)
write (u, "(1x,'[',A,']')") "kinematics set"
case (SF_FAILED_EVALUATION)
write (u, "(1x,'[',A,']')") "evaluation failed"
case (SF_EVALUATED)

```

```

        write (u, "(1x,['',A,'])") "evaluated"
    end select
end subroutine write_sf_status

```

Finalizer. Required because the base interaction needs a finalizer. We provide an implementation, which need not be overwritten if the implementation does not add finalizable subobjects.

```

<SF base: sf int: TBP>≡
    procedure :: final => sf_int_final

<SF base: procedures>+≡
    subroutine sf_int_final (object)
        class(sf_int_t), intent(inout) :: object
        call interaction_final (object%interaction_t)
    end subroutine sf_int_final

```

This is the basic output routine. Display status and interaction.

```

<SF base: sf int: TBP>+≡
    procedure :: base_write => sf_int_base_write

<SF base: procedures>+≡
    subroutine sf_int_base_write (object, unit)
        class(sf_int_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)", advance="no") "SF instance:"
        call write_sf_status (object%status, u)
        if (allocated (object%beam_index)) &
            write (u, "(3x,A,2(1x,I0))") "beam      =", object%beam_index
        if (allocated (object%incoming)) &
            write (u, "(3x,A,2(1x,I0))") "incoming  =", object%incoming
        if (allocated (object%radiated)) &
            write (u, "(3x,A,2(1x,I0))") "radiated  =", object%radiated
        if (allocated (object%outgoing)) &
            write (u, "(3x,A,2(1x,I0))") "outgoing  =", object%outgoing
        if (allocated (object%par_index)) &
            write (u, "(3x,A,2(1x,I0))") "parameter =", object%par_index
        if (object%qmin_defined) &
            write (u, "(3x,A,1x,ES19.12)") "q_min     =", object%qmin
        if (object%qmax_defined) &
            write (u, "(3x,A,1x,ES19.12)") "q_max     =", object%qmax
        call interaction_write (object%interaction_t, u)
    end subroutine sf_int_base_write

```

The type string identifies the structure function class, and possibly more details about the structure function.

```

<SF base: sf int: TBP>+≡
    procedure (sf_int_type_string), deferred :: type_string

<SF base: interfaces>+≡
    abstract interface
        function sf_int_type_string (object) result (string)

```

```

import
class(sf_int_t), intent(in) :: object
type(string_t) :: string
end function sf_int_type_string
end interface

```

Output of the concrete object. We should not forget to call the output routine for the base type.

```

<SF base: sf_int: TBP>+≡
  procedure (sf_int_write), deferred :: write

<SF base: interfaces>+≡
  abstract interface
    subroutine sf_int_write (object, unit)
      import
      class(sf_int_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine sf_int_write
  end interface

```

Basic initialization: set the invariant masses for the particles and initialize the interaction. The caller should then add states to the interaction and freeze it.

The dimension of the mask should be equal to the sum of the dimensions of the mass-squared arrays, which determine incoming, radiated, and outgoing particles, respectively.

Optionally, we can define minimum and maximum values for the momentum transfer to the outgoing particle(s). If all masses are zero, this is actually required for non-collinear splitting.

```

<SF base: sf_int: TBP>+≡
  procedure :: base_init => sf_int_base_init

<SF base: procedures>+≡
  subroutine sf_int_base_init &
    (sf_int, mask, mi2, mr2, mo2, qmin, qmax, hel_lock)
    class(sf_int_t), intent(out) :: sf_int
    type (quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    real(default), dimension(:), intent(in) :: mi2, mr2, mo2
    real(default), dimension(:), intent(in), optional :: qmin, qmax
    integer, dimension(:), intent(in), optional :: hel_lock
    allocate (sf_int%mi2 (size (mi2)))
    sf_int%mi2 = mi2
    allocate (sf_int%mr2 (size (mr2)))
    sf_int%mr2 = mr2
    allocate (sf_int%mo2 (size (mo2)))
    sf_int%mo2 = mo2
    if (present (qmin)) then
      sf_int%qmin_defined = .true.
      allocate (sf_int%qmin (size (qmin)))
      sf_int%qmin = qmin
    end if
    if (present (qmax)) then
      sf_int%qmax_defined = .true.
      allocate (sf_int%qmax (size (qmax)))

```



```

        sf_int%qmax = qmax
    end if
    call interaction_init (sf_int%interaction_t, &
        size (mi2), 0, size (mr2) + size (mo2), &
        mask = mask, hel_lock = hel_lock, set_relations = .true.)
end subroutine sf_int_base_init

```

Set the indices of the incoming, radiated, and outgoing particles, respectively.

```

<SF base: sf_int: TBP>+≡
    procedure :: set_incoming => sf_int_set_incoming
    procedure :: set_radiated => sf_int_set_radiated
    procedure :: set_outgoing => sf_int_set_outgoing

<SF base: procedures>+≡
    subroutine sf_int_set_incoming (sf_int, incoming)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: incoming
        allocate (sf_int%incoming (size (incoming)))
        sf_int%incoming = incoming
    end subroutine sf_int_set_incoming

    subroutine sf_int_set_radiated (sf_int, radiated)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: radiated
        allocate (sf_int%radiated (size (radiated)))
        sf_int%radiated = radiated
    end subroutine sf_int_set_radiated

    subroutine sf_int_set_outgoing (sf_int, outgoing)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: outgoing
        allocate (sf_int%outgoing (size (outgoing)))
        sf_int%outgoing = outgoing
    end subroutine sf_int_set_outgoing

```

Initialization. This proceeds via an abstract data object, which for the actual implementation should have the matching concrete type. Since all implementations have the same signature, we can prepare a deferred procedure. The data object will become the target of a corresponding pointer within the `sf_int_t` implementation.

This should call the previous procedure.

```

<SF base: sf_int: TBP>+≡
    procedure (sf_int_init), deferred :: init

<SF base: interfaces>+≡
    abstract interface
        subroutine sf_int_init (sf_int, data)
            import
            class(sf_int_t), intent(out) :: sf_int
            class(sf_data_t), intent(in), target :: data
        end subroutine sf_int_init
    end interface

```

Set beam indices, i.e., the beam(s) on which this structure function applies.

```

(SF base: sf_int: TBP)+≡
  procedure :: set_beam_index => sf_int_set_beam_index

(SF base: procedures)+≡
  subroutine sf_int_set_beam_index (sf_int, beam_index)
    class(sf_int_t), intent(inout) :: sf_int
    integer, dimension(:), intent(in) :: beam_index
    allocate (sf_int%beam_index (size (beam_index)))
    sf_int%beam_index = beam_index
  end subroutine sf_int_set_beam_index

```

Set parameter indices, indicating which MC input parameters are to be used for evaluating this structure function.

```

(SF base: sf_int: TBP)+≡
  procedure :: set_par_index => sf_int_set_par_index

(SF base: procedures)+≡
  subroutine sf_int_set_par_index (sf_int, par_index)
    class(sf_int_t), intent(inout) :: sf_int
    integer, dimension(:), intent(in) :: par_index
    allocate (sf_int%par_index (size (par_index)))
    sf_int%par_index = par_index
  end subroutine sf_int_set_par_index

```

Initialize the structure-function kinematics, setting incoming momenta. We assume that array shapes match.

Three versions. The first version relies on the momenta being linked to another interaction. The second version sets the momenta explicitly. In the third version, we first compute momenta for the specified energies and store those.

```

(SF base: sf_int: TBP)+≡
  generic :: seed_kinematics => sf_int_receive_momenta
  generic :: seed_kinematics => sf_int_seed_momenta
  generic :: seed_kinematics => sf_int_seed_energies
  procedure :: sf_int_receive_momenta
  procedure :: sf_int_seed_momenta
  procedure :: sf_int_seed_energies

(SF base: procedures)+≡
  subroutine sf_int_receive_momenta (sf_int)
    class(sf_int_t), intent(inout) :: sf_int
    if (sf_int%status >= SF_INITIAL) then
      call interaction_receive_momenta (sf_int%interaction_t)
      sf_int%status = SF_SEED_KINEMATICS
    end if
  end subroutine sf_int_receive_momenta

  subroutine sf_int_seed_momenta (sf_int, k)
    class(sf_int_t), intent(inout) :: sf_int
    type(vector4_t), dimension(:), intent(in) :: k
    if (sf_int%status >= SF_INITIAL) then
      call interaction_set_momenta (sf_int%interaction_t, k, &
        outgoing=.false.)
    end if
  end subroutine sf_int_seed_momenta

```

```

        sf_int%status = SF_SEED_KINEMATICS
    end if
end subroutine sf_int_seed_momenta

subroutine sf_int_seed_energies (sf_int, E)
    class(sf_int_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: E
    type(vector4_t), dimension(:), allocatable :: k
    integer :: j
    if (sf_int%status >= SF_INITIAL) then
        allocate (k (size (E)))
        if (all (E**2 >= sf_int%mi2)) then
            do j = 1, size (E)
                k(j) = vector4_moving (E(j), &
                    (3-2*j) * sqrt (E(j)**2 - sf_int%mi2(j)), 3)
            end do
            call sf_int%seed_kinematics (k)
        end if
    end if
end subroutine sf_int_seed_energies

```

Complete the structure-function kinematics, derived from an input parameter (array)  $r$  between 0 and 1. The interaction momenta are calculated, and we return  $x$  (the momentum fraction), and  $f$  (the Jacobian factor for the map  $r \rightarrow x$ ), if `map` is set.

If the `map` flag is unset,  $r$  and  $x$  values will coincide, and  $f$  will become unity. If it is set, the structure-function implementation chooses a convenient mapping from  $r$  to  $x$  with Jacobian  $f$ .

In the `inverse_kinematics` variant, we exchange the intent of `x` and `r`. The momenta are calculated only if the optional flag `set_momenta` is present and set.

*(SF base: sf\_int: TBP)+≡*

```

    procedure (sf_int_complete_kinematics), deferred :: complete_kinematics
    procedure (sf_int_inverse_kinematics), deferred :: inverse_kinematics

```

*(SF base: interfaces)+≡*

```

abstract interface
    subroutine sf_int_complete_kinematics (sf_int, x, f, r, map)
        import
        class(sf_int_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(out) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: r
        logical, intent(in) :: map
    end subroutine sf_int_complete_kinematics
end interface

abstract interface
    subroutine sf_int_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        import
        class(sf_int_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
    end subroutine sf_int_inverse_kinematics
end interface

```

```

        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
    end subroutine sf_int_inverse_kinematics
end interface

```

Single splitting: compute momenta, given  $x$  input parameters. We assume that the incoming momentum is set. The status code is set to `SF_FAILED_KINEMATICS` if the  $x$  array does not correspond to a valid momentum configuration. Otherwise, it is updated to `SF_DONE_KINEMATICS`.

We force the outgoing particle on-shell. The on-shell projection is determined by the `on_shell_mode`. The radiated particle should already be on shell.

```

<SF base: sf_int: TBP>+=
    procedure :: split_momentum => sf_int_split_momentum

<SF base: procedures>+=
    subroutine sf_int_split_momentum (sf_int, x, xb1)
        class(sf_int_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(in) :: xb1
        type(vector4_t) :: k
        type(vector4_t), dimension(2) :: q
        type(splitting_data_t) :: sd
        real(default) :: E1, E2
        logical :: fail
        if (sf_int%status >= SF_SEED_KINEMATICS) then
            k = interaction_get_momentum (sf_int%interaction_t, 1)
            call sd%init (k, &
                sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
                collinear = size (x) == 1)
            call sd%set_t_bounds (x(1), xb1)
            select case (size (x))
            case (1)
            case (3)
                if (sf_int%qmax_defined) then
                    if (sf_int%qmin_defined) then
                        call sd%sample_t (x(2), &
                            t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
                    else
                        call sd%sample_t (x(2), &
                            t0 = - sf_int%qmax(1) ** 2)
                    end if
                else
                    if (sf_int%qmin_defined) then
                        call sd%sample_t (x(2), t1 = - sf_int%qmin(1) ** 2)
                    else
                        call sd%sample_t (x(2))
                    end if
                end if
                call sd%sample_phi (x(3))
            case default
                call msg_bug ("Structure function: impossible number of parameters")
            end select
            q = sd%split_momentum (k)
            call on_shell (q, [sf_int%mr2, sf_int%mo2], &

```

```

        sf_int%on_shell_mode)
    call interaction_set_momenta (sf_int%interaction_t, &
        q, outgoing=.true.)
    E1 = energy (q(1))
    E2 = energy (q(2))
    fail = E1 < 0 .or. E2 < 0 &
        .or. E1 ** 2 < sf_int%mr2(1) &
        .or. E2 ** 2 < sf_int%mo2(1)
    if (fail) then
        sf_int%status = SF_FAILED_KINEMATICS
    else
        sf_int%status = SF_DONE_KINEMATICS
    end if
end if
end subroutine sf_int_split_momentum

```

Pair splitting: two incoming momenta, two radiated, two outgoing. This is simple because we insist on all momenta being collinear.

*(SF base: sf\_int: TBP)+≡*

```

    procedure :: split_momenta => sf_int_split_momenta

```

*(SF base: procedures)+≡*

```

subroutine sf_int_split_momenta (sf_int, x, xb1)
    class(sf_int_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), dimension(:), intent(in) :: xb1
    type(vector4_t), dimension(2) :: k
    type(vector4_t), dimension(4) :: q
    real(default), dimension(4) :: E
    logical :: fail
    if (sf_int%status >= SF_SEED_KINEMATICS) then
        select case (size (x))
        case (2)
        case default
            call msg_bug ("Pair structure function: recoil requested &
                &but not implemented yet")
        end select
        k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
        k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
        q(1:2) = xb1 * k
        q(3:4) = x * k
        call on_shell (q, &
            [sf_int%mr2(1), sf_int%mr2(2), &
            sf_int%mo2(1), sf_int%mo2(2)], &
            sf_int%on_shell_mode)
        call interaction_set_momenta (sf_int%interaction_t, &
            q, outgoing=.true.)
        E = energy (q)
        fail = any (E < 0) &
            .or. any (E(1:2) ** 2 < sf_int%mr2) &
            .or. any (E(3:4) ** 2 < sf_int%mo2)
        if (fail) then
            sf_int%status = SF_FAILED_KINEMATICS
        else

```

```

        sf_int%status = SF_DONE_KINEMATICS
    end if
end if
end subroutine sf_int_split_momenta

```

Pair spectrum: the reduced version of the previous splitting, without radiated momenta.

```

⟨SF base: sf_int: TBP⟩+=
    procedure :: reduce_momenta => sf_int_reduce_momenta

⟨SF base: procedures⟩+=
    subroutine sf_int_reduce_momenta (sf_int, x)
        class(sf_int_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        type(vector4_t), dimension(2) :: k
        type(vector4_t), dimension(2) :: q
        real(default), dimension(2) :: E
        logical :: fail
        if (sf_int%status >= SF_SEED_KINEMATICS) then
            select case (size (x))
            case (2)
            case default
                call msg_bug ("Pair spectrum: recoil requested &
                    &but not implemented yet")
            end select
            k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
            k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
            q = x * k
            call on_shell (q, &
                [sf_int%mo2(1), sf_int%mo2(2)], &
                sf_int%on_shell_mode)
            call interaction_set_momenta (sf_int%interaction_t, &
                q, outgoing=.true.)
            E = energy (q)
            fail = any (E < 0) &
                .or. any (E ** 2 < sf_int%mo2)
            if (fail) then
                sf_int%status = SF_FAILED_KINEMATICS
            else
                sf_int%status = SF_DONE_KINEMATICS
            end if
        end if
    end subroutine sf_int_reduce_momenta

```

The inverse procedure: we compute the x array from the momentum configuration.

NOTE: Here and above, the single-particle case is treated in detail, allowing for non-collinearity and non-vanishing masses and nontrivial momentum-transfer bounds. For the pair case, we currently implement only collinear splitting and assume massless particles. This should be improved.

```

⟨SF base: sf_int: TBP⟩+=
    procedure :: recover_x => sf_int_recover_x

```

*(SF base: procedures)* +=

```

subroutine sf_int_recover_x (sf_int, x)
  class(sf_int_t), intent(in) :: sf_int
  real(default), dimension(:), intent(out) :: x
  type(vector4_t), dimension(:), allocatable :: k
  type(vector4_t), dimension(:), allocatable :: q
  type(splitting_data_t) :: sd
  if (sf_int%status >= SF_SEED_KINEMATICS) then
    allocate (k (interaction_get_n_in (sf_int%interaction_t)))
    allocate (q (interaction_get_n_out (sf_int%interaction_t)))
    k = interaction_get_momenta (sf_int%interaction_t, outgoing=.false.)
    q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
    select case (size (k))
    case (1)
      call sd%init (k(1), &
        sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
        collinear = size (x) == 1)
      call sd%recover (k(1), q(2), sf_int%on_shell_mode)
      x(1) = sd%get_x ()
      select case (size (x))
      case (1)
      case (3)
        if (sf_int%qmax_defined) then
          if (sf_int%qmin_defined) then
            call sd%inverse_t (x(2), &
              t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
          else
            call sd%inverse_t (x(2), &
              t0 = - sf_int%qmax(1) ** 2)
          end if
        else
          if (sf_int%qmin_defined) then
            call sd%inverse_t (x(2), t1 = - sf_int%qmin(1) ** 2)
          else
            call sd%inverse_t (x(2))
          end if
        end if
        call sd%inverse_phi (x(3))
      case default
        call msg_bug ("Structure function: impossible number &
          &of parameters")
      end select
    case (2)
      select case (size (x))
      case (2)
      case default
        call msg_bug ("Pair structure function: recoil requested &
          &but not implemented yet")
      end select
    select case (sf_int%on_shell_mode)
    case (KEEP_ENERGY)
      select case (size (q))
      case (4)
        x = energy (q(3:4)) / energy (k)

```

```

        case (2)
            x = energy (q) / energy (k)
        end select
    case (KEEP_MOMENTUM)
        select case (size (q))
            case (4)
                x = longitudinal_part (q(3:4)) / longitudinal_part (k)
            case (2)
                x = longitudinal_part (q) / longitudinal_part (k)
            end select
        end select
    end select
end if
end subroutine sf_int_recover_x

```

Apply the structure function, i.e., evaluate the interaction. For the calculation, we may use the stored momenta, any further information stored inside the `sf_int` implementation during kinematics setup, and the given energy scale. It may happen that for the given kinematics the value is not defined. This should be indicated by the status code.

```

<SF base: sf_int: TBP>+≡
    procedure (sf_int_apply), deferred :: apply

<SF base: interfaces>+≡
    abstract interface
        subroutine sf_int_apply (sf_int, scale)
            import
            class(sf_int_t), intent(inout) :: sf_int
            real(default), intent(in) :: scale
        end subroutine sf_int_apply
    end interface

```

### 10.3.4 Accessing the structure function

Return metadata. Once `interaction_t` is rewritten in OO, some of this will be inherited.

The number of outgoing is equal to the number of incoming particles. The radiated particles are the difference.

```

<SF base: sf_int: TBP>+≡
    procedure :: get_n_in => sf_int_get_n_in
    procedure :: get_n_rad => sf_int_get_n_rad
    procedure :: get_n_out => sf_int_get_n_out

<SF base: procedures>+≡
    function sf_int_get_n_in (sf_int) result (n_in)
        class(sf_int_t), intent(in) :: sf_int
        integer :: n_in
        n_in = interaction_get_n_in (sf_int%interaction_t)
    end function sf_int_get_n_in

    function sf_int_get_n_rad (sf_int) result (n_rad)
        class(sf_int_t), intent(in) :: sf_int

```



```

integer :: n_rad
n_rad = interaction_get_n_out (sf_int%interaction_t) &
      - interaction_get_n_in (sf_int%interaction_t)
end function sf_int_get_n_rad

function sf_int_get_n_out (sf_int) result (n_out)
class(sf_int_t), intent(in) :: sf_int
integer :: n_out
n_out = interaction_get_n_in (sf_int%interaction_t)
end function sf_int_get_n_out

```

Number of matrix element entries in the interaction:

```

<SF base: sf_int: TBP>+≡
  procedure :: get_n_states => sf_int_get_n_states

<SF base: procedures>+≡
  function sf_int_get_n_states (sf_int) result (n_states)
class(sf_int_t), intent(in) :: sf_int
integer :: n_states
n_states = interaction_get_n_matrix_elements (sf_int%interaction_t)
end function sf_int_get_n_states

```

Return a specific state as a quantum-number array.

```

<SF base: sf_int: TBP>+≡
  procedure :: get_state => sf_int_get_state

<SF base: procedures>+≡
  function sf_int_get_state (sf_int, i) result (qn)
class(sf_int_t), intent(in) :: sf_int
type(quantum_numbers_t), dimension(:), allocatable :: qn
integer, intent(in) :: i
allocate (qn (interaction_get_n_tot (sf_int%interaction_t)))
qn = interaction_get_quantum_numbers (sf_int%interaction_t, i)
end function sf_int_get_state

```

Return the matrix-element values for all states. We can assume that the matrix elements are real, so we take the real part.

```

<SF base: sf_int: TBP>+≡
  procedure :: get_values => sf_int_get_values

<SF base: procedures>+≡
  subroutine sf_int_get_values (sf_int, value)
class(sf_int_t), intent(in) :: sf_int
real(default), dimension(:), intent(out) :: value
integer :: i
if (sf_int%status >= SF_EVALUATED) then
  do i = 1, size (value)
    value(i) = interaction_get_matrix_element &
      (sf_int%interaction_t, i)
  end do
else
  value = 0
end if

```

```
end subroutine sf_int_get_values
```

### 10.3.5 Direct calculations

Compute a structure function value (array) directly, given an array of  $x$  values and a scale. If the energy is also given, we initialize the kinematics for that energy, otherwise take it from a previous run.

We assume that the  $E$  array has dimension  $n\_in$ , and the  $x$  array has  $n\_par$ .

```
(SF base: sf_int: TBP)+≡
  procedure :: compute_values => sf_int_compute_values

(SF base: procedures)+≡
  subroutine sf_int_compute_values (sf_int, value, x, scale, E)
    class(sf_int_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: value
    real(default), dimension(:), intent(in) :: x
    real(default), intent(in) :: scale
    real(default), dimension(:), intent(in), optional :: E
    real(default), dimension(size (x)) :: xx
    real(default) :: f
    if (present (E)) call sf_int%seed_kinematics (E)
    if (sf_int%status >= SF_SEED_KINEMATICS) then
      call sf_int%complete_kinematics (xx, f, x, map=.false.)
      call sf_int%apply (scale)
      call sf_int%get_values (value)
      value = value * f
    else
      value = 0
    end if
  end subroutine sf_int_compute_values
```

Compute just a single value for one of the states, i.e., throw the others away.

```
(SF base: sf_int: TBP)+≡
  procedure :: compute_value => sf_int_compute_value

(SF base: procedures)+≡
  subroutine sf_int_compute_value &
    (sf_int, i_state, value, x, scale, E)
    class(sf_int_t), intent(inout) :: sf_int
    integer, intent(in) :: i_state
    real(default), intent(out) :: value
    real(default), dimension(:), intent(in) :: x
    real(default), intent(in) :: scale
    real(default), dimension(:), intent(in), optional :: E
    real(default), dimension(:), allocatable :: value_array
    if (sf_int%status >= SF_INITIAL) then
      allocate (value_array (sf_int%get_n_states ()))
      call sf_int%compute_values (value_array, x, scale, E)
      value = value_array(i_state)
    else
      value = 0
    end if
```

```
end subroutine sf_int_compute_value
```

### 10.3.6 Structure-function instance

This is a wrapper for `sf_int_t` objects, such that we can build an array with different structure-function types. The structure-function contains an array (a sequence) of `sf_int_t` objects.

The object, it holds the evaluator that connects the preceding part of the structure-function chain to the current interaction.

It also stores the input and output parameter values for the contained structure function. The `r` array has a second dimension, corresponding to the mapping channels in a multi-channel configuration. There is a Jacobian entry `f` for each channel. The corresponding logical array `mapping` tells whether we apply the mapping appropriate for the current structure function in this channel. The `x` parameter values (energy fractions) are common to all channels.

*<SF base: types>+≡*

```
type :: sf_instance_t
  class(sf_int_t), allocatable :: int
  type(evaluator_t) :: eval
  real(default), dimension(:,:), allocatable :: r
  real(default), dimension(:), allocatable :: f
  logical, dimension(:), allocatable :: m
  real(default), dimension(:), allocatable :: x
end type sf_instance_t
```

### 10.3.7 Structure-function chain

A chain is an array of structure functions `sf`, initiated by a beam setup. We do not use this directly for evaluation, but create instances with copies of the contained interactions.

`n_par` is the total number of MC input parameters that is necessary for completely determining the structure-function chain.

*<SF base: public>+≡*

```
public :: sf_chain_t
```

*<SF base: types>+≡*

```
type, extends (beam_t) :: sf_chain_t
  type(beam_data_t), pointer :: beam_data => null ()
  integer :: n_in = 0
  integer :: n_strfun = 0
  integer :: n_par = 0
  type(sf_instance_t), dimension(:), allocatable :: sf
  contains
    <SF base: sf chain: TBP>
end type sf_chain_t
```

Finalizer.

*<SF base: sf chain: TBP>≡*

```
procedure :: final => sf_chain_final
```

```

<SF base: procedures>+=
subroutine sf_chain_final (object)
class(sf_chain_t), intent(inout) :: object
integer :: i
if (allocated (object%sf)) then
do i = 1, size (object%sf, 1)
associate (sf => object%sf(i))
if (allocated (sf%int)) then
call sf%int%final ()
end if
end associate
end do
end if
call beam_final (object%beam_t)
end subroutine sf_chain_final

```

Output.

```

<SF base: sf chain: TBP>+=
procedure :: write => sf_chain_write
<SF base: procedures>+=
subroutine sf_chain_write (object, unit)
class(sf_chain_t), intent(in) :: object
integer, intent(in), optional :: unit
integer :: u, i, c
u = output_unit (unit)
write (u, "(1x,A)") "Incoming particles / structure-function chain:"
if (associated (object%beam_data)) then
write (u, "(3x,A,I0)") "n_in      = ", object%n_in
write (u, "(3x,A,I0)") "n_strfun  = ", object%n_strfun
write (u, "(3x,A,I0)") "n_par      = ", object%n_par
call beam_data_write (object%beam_data, u)
call write_separator (u)
call beam_write (object%beam_t, u)
if (allocated (object%sf)) then
do i = 1, object%n_strfun
associate (sf => object%sf(i))
call write_separator (u)
if (allocated (sf%int)) then
call sf%int%write (u)
else
write (u, "(1x,A)") "SF instance: [undefined]"
end if
end associate
end do
end if
else
write (u, "(3x,A)") "[undefined]"
end if
end subroutine sf_chain_write

```

Initialize: setup beams. The `beam_data` target must remain valid for the lifetime of the chain, since we just establish a pointer. The structure-function configuration array is used to initialize the individual structure-function entries. The

target attribute is needed because the `sf_int` entries establish pointers to the configuration data.

```

<SF base: sf chain: TBP>+≡
  procedure :: init => sf_chain_init

<SF base: procedures>+≡
  subroutine sf_chain_init (sf_chain, beam_data, sf_config)
    class(sf_chain_t), intent(out) :: sf_chain
    type(beam_data_t), intent(in), target :: beam_data
    type(sf_config_t), dimension(:), intent(in), optional, target :: sf_config
    integer :: i
    sf_chain%beam_data => beam_data
    sf_chain%n_in = beam_data_get_n_in (beam_data)
    call beam_init (sf_chain%beam_t, beam_data)
    if (present (sf_config)) then
      sf_chain%n_strfun = size (sf_config)
      allocate (sf_chain%sf (sf_chain%n_strfun))
      do i = 1, sf_chain%n_strfun
        call sf_chain%set_strfun (i, sf_config(i)%i, sf_config(i)%data)
      end do
    end if
  end subroutine sf_chain_init

```

Set a structure-function entry. We use the `data` input to allocate the `int` structure-function instance with appropriate type, then initialize the entry. The entry establishes a pointer to `data`.

The index `i` is the structure-function index in the chain.

```

<SF base: sf chain: TBP>+≡
  procedure :: set_strfun => sf_chain_set_strfun

<SF base: procedures>+≡
  subroutine sf_chain_set_strfun (sf_chain, i, beam_index, data)
    class(sf_chain_t), intent(inout) :: sf_chain
    integer, intent(in) :: i
    integer, dimension(:), intent(in) :: beam_index
    class(sf_data_t), intent(in), target :: data
    integer :: n_par, j
    n_par = data%get_n_par ()
    call data%allocate_sf_int (sf_chain%sf(i)%int)
    associate (sf_int => sf_chain%sf(i)%int)
      call sf_int%init(data)
      call sf_int%set_beam_index (beam_index)
      call sf_int%set_par_index &
        ([j, j = sf_chain%n_par + 1, sf_chain%n_par + n_par])
    end associate
    sf_chain%n_par = sf_chain%n_par + n_par
  end subroutine sf_chain_set_strfun

```

Return the number of structure-function parameters.

```

<SF base: sf chain: TBP>+≡
  procedure :: get_n_par => sf_chain_get_n_par

```

```

<SF base: procedures>+=
function sf_chain_get_n_par (sf_chain) result (n)
  class(sf_chain_t), intent(in) :: sf_chain
  integer :: n
  n = sf_chain%n_par
end function sf_chain_get_n_par

```

### 10.3.8 Auxiliary stuff

Write a separator line.

```

<SF base: procedures>+=
subroutine write_separator (u)
  integer, intent(in) :: u
  write (u, "(A)") repeat (" ", 72)
end subroutine write_separator

subroutine write_separator_double (u)
  integer, intent(in) :: u
  write (u, "(A)") repeat ("=", 72)
end subroutine write_separator_double

```

### 10.3.9 Chain instances

A structure-function chain instance contains copies of the interactions in the configuration chain, suitably linked to each other and connected by evaluators.

After initialization, `out_sf` should point, for each beam, to the last structure function that affects this beam. `out_sf_i` should indicate the index of the corresponding outgoing particle within that structure-function interaction.

Analogously, `out_eval` is the last evaluator in the structure-function chain, which contains the complete set of outgoing particles. `out_eval_i` should indicate the index of the outgoing particles, within that evaluator, which will initiate the collision.

When calculating actual kinematics, we fill the `p`, `r`, and `x` arrays and the `f` factor. The `p` array denotes the MC input parameters as they come from the random-number generator. The `r` array results from applying global mappings. The `x` array results from applying structure-function local mappings. The `x` values can be interpreted directly as momentum fractions (or angle fractions, where recoil is involved). The `f` factor is the Jacobian that results from applying all mappings.

The `mapping` entry may store a global mapping that is applied to a combination of `x` values and structure functions, as opposed to mappings that affect only a single structure function. It is applied before the latter mappings, in the transformation from the `p` array to the `r` array. For parameters affected by this mapping, we should ensure that they are not involved in a local mapping.

```

<SF base: public>+=
public :: sf_chain_instance_t

```

```

<SF base: types>+≡
type, extends (beam_t) :: sf_chain_instance_t
  type(sf_chain_t), pointer :: config => null ()
  integer :: status = SF_UNDEFINED
  type(sf_instance_t), dimension(:), allocatable :: sf
  integer, dimension(:), allocatable :: out_sf
  integer, dimension(:), allocatable :: out_sf_i
  integer :: out_eval = 0
  integer, dimension(:), allocatable :: out_eval_i
  integer :: selected_channel = 0
  real(default), dimension(:,:), allocatable :: p
  real(default), dimension(:,:), allocatable :: r
  real(default), dimension(:), allocatable :: f
  real(default), dimension(:), allocatable :: x
  type(sf_channel_t), dimension(:), allocatable :: channel
contains
  <SF base: sf chain instance: TBP>
end type sf_chain_instance_t

```

Finalizer.

```

<SF base: sf chain instance: TBP>≡
  procedure :: final => sf_chain_instance_final

<SF base: procedures>+≡
  subroutine sf_chain_instance_final (object)
    class(sf_chain_instance_t), intent(inout) :: object
    integer :: i
    if (allocated (object%sf)) then
      do i = 1, size (object%sf, 1)
        associate (sf => object%sf(i))
          if (allocated (sf%int)) then
            call evaluator_final (sf%eval)
            call sf%int%final ()
          end if
        end associate
      end do
    end if
    call beam_final (object%beam_t)
  end subroutine sf_chain_instance_final

```

Output.

Note: nagfor 5.3.1 appears to be slightly confused with the allocation status. We check both for allocation and nonzero size.

```

<SF base: sf chain instance: TBP>+≡
  procedure :: write => sf_chain_instance_write

<SF base: procedures>+≡
  subroutine sf_chain_instance_write (object, unit)
    class(sf_chain_instance_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i, c
    u = output_unit (unit)
    write (u, "(1x,A)", advance="no") "Structure-function chain instance:"
    call write_sf_status (object%status, u)

```

```

if (allocated (object%out_sf)) then
  write (u, "(3x,A)", advance="no") "outgoing (interactions) ="
  do i = 1, size (object%out_sf)
    write (u, "(1x,I0,':',I0)", advance="no") &
      object%out_sf(i), object%out_sf_i(i)
  end do
  write (u, *)
end if
if (object%out_eval /= 0) then
  write (u, "(3x,A)", advance="no") "outgoing (evaluators) ="
  do i = 1, size (object%out_sf)
    write (u, "(1x,I0,':',I0)", advance="no") &
      object%out_eval, object%out_eval_i(i)
  end do
  write (u, *)
end if
if (allocated (object%sf)) then
  if (size (object%sf) /= 0) then
    write (u, "(1x,A)") "Structure-function parameters:"
    do c = 1, size (object%f)
      write (u, "(1x,A,I0,A)", advance="no") "Channel #", c, ":"
      if (c == object%selected_channel) then
        write (u, "(1x,A)") "[selected]"
      else
        write (u, *)
      end if
      write (u, "(3x,A,9(1x,F9.7))") "p =", object%p(:,c)
      write (u, "(3x,A,9(1x,F9.7))") "r =", object%r(:,c)
      write (u, "(3x,A,9(1x,ES13.7))") "f =", object%f(c)
      write (u, "(3x,A)", advance="no") "m ="
      call object%channel(c)%write (u)
    end do
    write (u, "(3x,A,9(1x,F9.7))") "x =", object%x
  end if
end if
call write_separator (u)
call beam_write (object%beam_t, u)
if (allocated (object%sf)) then
  do i = 1, size (object%sf)
    associate (sf => object%sf(i))
    call write_separator (u)
    if (allocated (sf%int)) then
      if (allocated (sf%r)) then
        write (u, "(1x,A)") "Structure-function parameters:"
        do c = 1, size (sf%f)
          write (u, "(1x,A,I0,A)", advance="no") "Channel #", c, ":"
          if (c == object%selected_channel) then
            write (u, "(1x,A)") "[selected]"
          else
            write (u, *)
          end if
          write (u, "(3x,A,9(1x,F9.7))") "r =", sf%r(:,c)
          write (u, "(3x,A,9(1x,ES13.7))") "f =", sf%f(c)
          write (u, "(3x,A,9(1x,L1,7x))") "m =", sf%m(c)
        end do
      end if
    end if
  end do
end if

```



```

        end do
        write (u, "(3x,A,9(1x,F9.7))" "x =", sf%x
    end if
    call sf%int%write (u)
    if (.not. evaluator_is_empty (sf%eval)) then
        call evaluator_write (sf%eval, u)
    end if
end if
end if
end associate
end do
end if
end subroutine sf_chain_instance_write

```

Initialize. This creates a copy of the interactions in the configuration chain, assumed to be properly initialized. In the copy, we allocate the `p` etc. arrays.

The brute-force assignment of the `sf` component would be straightforward, but at least gfortran 4.6.3 would like a more fine-grained copy. In any case, the copy is deep as far as allocatables are concerned, but for the contained `interaction_t` objects the copy is shallow, as long as we do not bind defined assignment to the type. Therefore, we have to re-assign the `interaction_t` components explicitly, this time calling the proper defined assignment. Furthermore, we allocate the parameter arrays for each structure function.

*(SF base: sf chain instance: TBP)+≡*

```

    procedure :: init => sf_chain_instance_init

```

*(SF base: procedures)+≡*

```

subroutine sf_chain_instance_init (chain, config, n_channel)
    class(sf_chain_instance_t), intent(out), target :: chain
    type(sf_chain_t), intent(in), target :: config
    integer, intent(in) :: n_channel
    integer :: i, c
    integer :: n_par_tot, n_par, n_strfun
    chain%config => config
    n_strfun = config%n_strfun
    chain%beam_t = config%beam_t
    allocate (chain%out_sf (config%n_in), chain%out_sf_i (config%n_in))
    allocate (chain%out_eval_i (config%n_in))
    chain%out_sf = 0
    chain%out_sf_i = [(i, i = 1, config%n_in)]
    chain%out_eval_i = chain%out_sf_i
    if (n_strfun /= 0) then
        n_par_tot = 0
        allocate (chain%sf (n_strfun))
        do i = 1, n_strfun
            associate (sf => chain%sf(i))
                allocate (sf%int, source=config%sf(i)%int)
                sf%int%interaction_t = config%sf(i)%int%interaction_t
                n_par = size (sf%int%par_index)
                allocate (sf%r (n_par, n_channel)); sf%r = 0
                allocate (sf%f (n_channel)); sf%f = 0
                allocate (sf%m (n_channel)); sf%m = .false.
                allocate (sf%x (n_par)); sf%x = 0
                n_par_tot = n_par_tot + n_par
            end associate
        end do
    end if
end subroutine

```

```

        end associate
    end do
    allocate (chain%p (n_par_tot, n_channel)); chain%p = 0
    allocate (chain%r (n_par_tot, n_channel)); chain%r = 0
    allocate (chain%f (n_channel)); chain%f = 0
    allocate (chain%x (n_par_tot)); chain%x = 0
    call allocate_sf_channels &
        (chain%channel, n_channel=n_channel, n_strfun=n_strfun)
    end if
    chain%status = SF_INITIAL
end subroutine sf_chain_instance_init

```

Manually select a channel.

```

<SF base: sf chain instance: TBP>+≡
    procedure :: select_channel => sf_chain_instance_select_channel

<SF base: procedures>+≡
    subroutine sf_chain_instance_select_channel (chain, channel)
        class(sf_chain_instance_t), intent(inout) :: chain
        integer, intent(in), optional :: channel
        if (present (channel)) then
            chain%selected_channel = channel
        else
            chain%selected_channel = 0
        end if
    end subroutine sf_chain_instance_select_channel

```

Copy a channel-mapping object to the structure-function chain instance. We assume that assignment is sufficient, i.e., any non-static components of the **channel** object are allocatable and thus recursively copied.

After the copy, we extract the single-entry mappings and activate them for the individual structure functions. If there is a multi-entry mapping, we obtain the corresponding MC parameter indices and set them in the copy of the channel object.

```

<SF base: sf chain instance: TBP>+≡
    procedure :: set_channel => sf_chain_instance_set_channel

<SF base: procedures>+≡
    subroutine sf_chain_instance_set_channel (chain, c, channel)
        class(sf_chain_instance_t), intent(inout) :: chain
        integer, intent(in) :: c
        type(sf_channel_t), intent(in) :: channel
        integer :: i, j
        if (chain%status >= SF_INITIAL) then
            chain%channel(c) = channel
            j = 0
            do i = 1, chain%config%n_strfun
                associate (sf => chain%sf(i))
                    sf%m(c) = channel%is_single_mapping (i)
                    if (channel%is_multi_mapping (i)) then
                        j = j + 1
                        call chain%channel(c)%set_par_index (j, sf%int%par_index(1))
                    end if
                end associate
            end do
        end if
    end subroutine sf_chain_instance_set_channel

```

```

        end associate
    end do
    chain%status = SF_INITIAL
end if
end subroutine sf_chain_instance_set_channel

```

Link the interactions in the chain. First, link the beam instance to its template in the configuration chain, which should have the appropriate momenta fixed.

Then, we follow the chain via the arrays `out_sf` and `out_sf_i`. The arrays are (up to) two-dimensional, the entries correspond to the beam particle(s). For each beam, the entry `out_sf` points to the last interaction that affected this beam, and `out_sf_i` is the out-particle index within that interaction. For the initial beam, `out_sf` is zero by definition.

For each entry in the chain, we scan the affected beams (one or two). We look for `out_sf` and link the out-particle there to the corresponding in-particle in the current interaction. Then, we update the entry in `out_sf` and `out_sf_i` to point to the current interaction.

```

<SF base: sf chain instance: TBP>+≡
    procedure :: link_interactions => sf_chain_instance_link_interactions

<SF base: procedures>+≡
    subroutine sf_chain_instance_link_interactions (chain)
        class(sf_chain_instance_t), intent(inout), target :: chain
        type(interaction_t), pointer :: int
        integer :: i, j, b
        if (chain%status >= SF_INITIAL) then
            do b = 1, chain%config%n_in
                int => beam_get_int_ptr (chain%beam_t)
                call interaction_set_source_link (int, b, &
                    chain%config%beam_t, b)
            end do
            if (allocated (chain%sf)) then
                do i = 1, size (chain%sf)
                    associate (sf_int => chain%sf(i)%int)
                        do j = 1, size (sf_int%beam_index)
                            b = sf_int%beam_index(j)
                            call link (sf_int%interaction_t, b, sf_int%incoming(j))
                            chain%out_sf(b) = i
                            chain%out_sf_i(b) = sf_int%outgoing(j)
                        end do
                    end associate
                end do
            end if
            chain%status = SF_DONE_LINKS
        end if
contains
        subroutine link (int, b, in_index)
            type(interaction_t), intent(inout) :: int
            integer, intent(in) :: b, in_index
            integer :: i
            i = chain%out_sf(b)
            select case (i)
            case (0)

```

```

        call interaction_set_source_link (int, in_index, &
            chain%beam_t, chain%out_sf_i(b))
    case default
        call interaction_set_source_link (int, in_index, &
            chain%sf(i)%int%interaction_t, chain%out_sf_i(b))
    end select
end subroutine link
end subroutine sf_chain_instance_link_interactions

```

Exchange the quantum-number masks between the interactions in the chain, so we can combine redundant entries and detect any obvious mismatch.

We proceed first in the forward direction and then backwards again.

*<SF base: sf chain instance: TBP>+≡*

```

    procedure :: exchange_mask => sf_chain_exchange_mask

```

*<SF base: procedures>+≡*

```

subroutine sf_chain_exchange_mask (chain)
    class(sf_chain_instance_t), intent(inout), target :: chain
    type(interaction_t), pointer :: int
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
    integer :: i
    if (chain%status >= SF_DONE_LINKS) then
        if (allocated (chain%sf)) then
            int => beam_get_int_ptr (chain%beam_t)
            allocate (mask (interaction_get_n_out (int)))
            mask = interaction_get_mask (int)
            if (size (chain%sf) /= 0) then
                do i = 1, size (chain%sf) - 1
                    call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
                end do
                do i = size (chain%sf), 1, -1
                    call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
                end do
                if (any (mask .neqv. interaction_get_mask (int))) then
                    chain%status = SF_FAILED_MASK
                    return
                end if
            end if
        end if
        chain%status = SF_DONE_MASK
    end if
end subroutine sf_chain_exchange_mask

```

Initialize the evaluators that connect the interactions in the chain.

*<SF base: sf chain instance: TBP>+≡*

```

    procedure :: init_evaluators => sf_chain_instance_init_evaluators

```

*<SF base: procedures>+≡*

```

subroutine sf_chain_instance_init_evaluators (chain)
    class(sf_chain_instance_t), intent(inout), target :: chain
    type(interaction_t), pointer :: int
    type(quantum_numbers_mask_t) :: mask
    integer :: i
    if (chain%status >= SF_DONE_MASK) then

```

```

    if (allocated (chain%sf)) then
    if (size (chain%sf) /= 0) then
        mask = new_quantum_numbers_mask (.false., .false., .true.)
        int => beam_get_int_ptr (chain%beam_t)
        do i = 1, size (chain%sf)
            associate (sf => chain%sf(i))
                call evaluator_init_product (sf%eval, &
                    int, sf%int%interaction_t, &
                    mask)
                if (evaluator_is_empty (sf%eval)) then
                    chain%status = SF_FAILED_CONNECTIONS
                    return
                end if
                int => evaluator_get_int_ptr (sf%eval)
            end associate
        end do
        call find_outgoing_particles ()
    end if
    end if
    chain%status = SF_DONE_CONNECTIONS
end if
contains
<SF base: init evaluators: find outgoing particles>
end subroutine sf_chain_instance_init_evaluators

```

This is an internal subroutine of the previous one: After evaluators are set, trace the outgoing particles to the last evaluator. We only need the first channel, all channels are equivalent for this purpose.

For each beam, the outgoing particle is located by `out_sf` (the structure-function object where it originates) and `out_sf_i` (the index within that object). This particle is referenced by the corresponding evaluator, which in turn is referenced by the next evaluator, until we are at the end of the chain. We can trace back references by `interaction_find_link`. Knowing that `out_eval` is the index of the last evaluator, we thus determine `out_eval_i`, the index of the outgoing particle within that evaluator.

```

<SF base: init evaluators: find outgoing particles>≡
subroutine find_outgoing_particles ()
    type(interaction_t), pointer :: int, int_next
    integer :: i, j, out_sf, out_i
    chain%out_eval = size (chain%sf)
    do j = 1, size (chain%out_eval_i)
        out_sf = chain%out_sf(j)
        out_i = chain%out_sf_i(j)
        if (out_sf == 0) then
            int => beam_get_int_ptr (chain%beam_t)
            out_sf = 1
        else
            int => chain%sf(out_sf)%int%interaction_t
        end if
    do i = out_sf, chain%out_eval
        int_next => evaluator_get_int_ptr (chain%sf(i)%eval)
        out_i = interaction_find_link (int_next, int, out_i)
        int => int_next
    end do
end subroutine

```

```

        end do
        chain%out_eval_i(j) = out_i
    end do
end subroutine find_outgoing_particles

```

Compute the kinematics in the chain instance. We can assume that the seed momenta are set in the configuration beams. Scanning the chain, we first transfer the incoming momenta. Then, we use up the MC input parameter array **p** to compute the radiated and outgoing momenta.

In the multi-channel case, **c\_sel** is the channel which we use for computing the kinematics and the **x** values. In the other channels, we invert the kinematics in order to recover the corresponding rows in the **r** array, and the Jacobian **f**.

We first apply any global mapping to transform the input **p** into the array **r**. This is then given to the structure functions which compute the final array **x** and Jacobian factors **f**, which we multiply to obtain the overall Jacobian.

*(SF base: sf chain instance: TBP)+≡*

```

    procedure :: compute_kinematics => sf_chain_instance_compute_kinematics

```

*(SF base: procedures)+≡*

```

subroutine sf_chain_instance_compute_kinematics (chain, c_sel, p_in)
    class(sf_chain_instance_t), intent(inout), target :: chain
    integer, intent(in) :: c_sel
    real(default), dimension(:), intent(in) :: p_in
    type(interaction_t), pointer :: int
    real(default) :: f_mapping
    integer :: i, j, c
    if (chain%status >= SF_DONE_CONNECTIONS) then
        call chain%select_channel (c_sel)
        int => beam_get_int_ptr (chain%beam_t)
        call interaction_receive_momenta (int)
        if (allocated (chain%sf)) then
            if (size (chain%sf) /= 0) then
                forall (i = 1:size (chain%sf)) chain%sf(i)%int%status = SF_INITIAL
                chain%p(:,c_sel) = p_in
                chain%f = 1
                if (allocated (chain%channel(c_sel)%multi_mapping)) then
                    call chain%channel(c_sel)%multi_mapping%compute &
                        (chain%r(:,c_sel), f_mapping, chain%p(:,c_sel))
                    chain%f(c_sel) = f_mapping
                else
                    chain%r(:,c_sel) = chain%p(:,c_sel)
                    chain%f(c_sel) = 1
                end if
            end if
            do i = 1, size (chain%sf)
                associate (sf => chain%sf(i))
                    call sf%int%seed_kinematics ()
                    do j = 1, size (sf%x)
                        sf%r(j,c_sel) = chain%r(sf%int%par_index(j),c_sel)
                    end do
                    call sf%int%complete_kinematics &
                        (sf%x, sf%f(c_sel), sf%r(:,c_sel), sf%m(c_sel))
                    do j = 1, size (sf%x)
                        chain%x(sf%int%par_index(j)) = sf%x(j)
                    end do
                end associate
            end do
        end if
    end if
end subroutine

```

```

        if (sf%int%status <= SF_FAILED_KINEMATICS) then
            chain%status = SF_FAILED_KINEMATICS
            return
        end if
        do c = 1, size (sf%f)
            if (c /= c_sel) then
                call sf%int%inverse_kinematics &
                    (sf%x, sf%f(c), sf%r(:,c), sf%m(c))
                do j = 1, size (sf%x)
                    chain%r(sf%int%par_index(j),c) = sf%r(j,c)
                end do
            end if
            chain%f(c) = chain%f(c) * sf%f(c)
        end do
        if (.not. evaluator_is_empty (sf%eval)) then
            call evaluator_receive_momenta (sf%eval)
        end if
    end associate
end do
do c = 1, size (chain%f)
    if (c /= c_sel) then
        if (allocated (chain%channel(c)%multi_mapping)) then
            call chain%channel(c)%multi_mapping%inverse &
                (chain%r(:,c), f_mapping, chain%p(:,c))
            chain%f(c) = chain%f(c) * f_mapping
        else
            chain%p(:,c) = chain%r(:,c)
        end if
    end if
end do
end if
end if
chain%status = SF_DONE_KINEMATICS
end if
end subroutine sf_chain_instance_compute_kinematics

```

This is a variant of the previous procedure. We know the  $x$  parameters and reconstruct the momenta and the MC input parameters  $p$ . We do not need to select a channel.

Note: this is probably redundant, since the method we actually want starts from the momenta, recovers all  $x$  parameters, and then inverts mappings. See below `recover_kinematics`.

*(SF base: sf chain instance: TBP)+≡*

```

    procedure :: inverse_kinematics => sf_chain_instance_inverse_kinematics

```

*(SF base: procedures)+≡*

```

    subroutine sf_chain_instance_inverse_kinematics (chain, x)
        class(sf_chain_instance_t), intent(inout), target :: chain
        real(default), dimension(:), intent(in) :: x
        type(interaction_t), pointer :: int
        real(default) :: f_mapping
        integer :: i, j, c
        if (chain%status >= SF_DONE_CONNECTIONS) then
            call chain%select_channel ()

```

```

int => beam_get_int_ptr (chain%beam_t)
call interaction_receive_momenta (int)
if (allocated (chain%sf)) then
  chain%f = 1
  if (size (chain%sf) /= 0) then
    forall (i = 1:size (chain%sf)) chain%sf(i)%int%status = SF_INITIAL
    chain%x = x
    do i = 1, size (chain%sf)
      associate (sf => chain%sf(i))
        call sf%int%seed_kinematics ()
        do j = 1, size (sf%x)
          sf%x(j) = chain%x(sf%int%par_index(j))
        end do
        do c = 1, size (sf%f)
          call sf%int%inverse_kinematics &
            (sf%x, sf%f(c), sf%r(:,c), sf%m(c), c==1)
          chain%f(c) = chain%f(c) * sf%f(c)
          do j = 1, size (sf%x)
            chain%r(sf%int%par_index(j),c) = sf%r(j,c)
          end do
        end do
      end do
      if (.not. evaluator_is_empty (sf%eval)) then
        call evaluator_receive_momenta (sf%eval)
      end if
    end associate
  end do
  do c = 1, size (chain%f)
    if (allocated (chain%channel(c)%multi_mapping)) then
      call chain%channel(c)%multi_mapping%inverse &
        (chain%r(:,c), f_mapping, chain%p(:,c))
      chain%f(c) = chain%f(c) * f_mapping
    else
      chain%p(:,c) = chain%r(:,c)
    end if
  end do
end if
chain%status = SF_DONE_KINEMATICS
end if
end subroutine sf_chain_instance_inverse_kinematics

```

Recover the kinematics: assuming that the last evaluator has been filled with a valid set of momenta, we travel the momentum links backwards and fill the preceding evaluators and, as a side effect, interactions. We stop at the beam interaction.

After all momenta are set, apply the `inverse_kinematics` procedure above, suitably modified, to recover the  $x$  and  $p$  parameters and the Jacobian factors.

The `c_sel` (channel) argument is just used to mark a selected channel for the records, otherwise the recovery procedure is independent of this.

*(SF base: sf chain instance: TBP)+≡*

```
procedure :: recover_kinematics => sf_chain_instance_recover_kinematics
```

*(SF base: procedures)+≡*



```

subroutine sf_chain_instance_recover_kinematics (chain, c_sel)
  class(sf_chain_instance_t), intent(inout), target :: chain
  integer, intent(in) :: c_sel
  real(default) :: f_mapping
  integer :: i, j, c
  if (chain%status >= SF_DONE_CONNECTIONS) then
    call chain%select_channel (c_sel)
    if (allocated (chain%sf)) then
      do i = size (chain%sf), 1, -1
        associate (sf => chain%sf(i))
          if (.not. evaluator_is_empty (sf%eval)) then
            call evaluator_send_momenta (sf%eval)
          end if
        end associate
      end do
      chain%f = 1
      if (size (chain%sf) /= 0) then
        forall (i = 1:size (chain%sf)) chain%sf(i)%int%status = SF_INITIAL
        do i = 1, size (chain%sf)
          associate (sf => chain%sf(i))
            call sf%int%seed_kinematics ()
            call sf%int%recover_x (sf%x)
            do j = 1, size (sf%x)
              chain%x(sf%int%par_index(j)) = sf%x(j)
            end do
            do c = 1, size (sf%f)
              call sf%int%inverse_kinematics &
                (sf%x, sf%f(c), sf%r(:,c), sf%m(c), c==1)
              chain%f(c) = chain%f(c) * sf%f(c)
              do j = 1, size (sf%x)
                chain%r(sf%int%par_index(j),c) = sf%r(j,c)
              end do
            end do
          end associate
        end do
        do c = 1, size (chain%f)
          if (allocated (chain%channel(c)%multi_mapping)) then
            call chain%channel(c)%multi_mapping%inverse &
              (chain%r(:,c), f_mapping, chain%p(:,c))
            chain%f(c) = chain%f(c) * f_mapping
          else
            chain%p(:,c) = chain%r(:,c)
          end if
        end do
      end if
      chain%status = SF_DONE_KINEMATICS
    end if
  end subroutine sf_chain_instance_recover_kinematics

```

Evaluate all interactions in the chain and the product evaluators. We provide a `scale` argument that is given to all structure functions in the chain.

*(SF base: sf chain instance: TBP)+≡*

```

    procedure :: evaluate => sf_chain_instance_evaluate
  (SF base: procedures)+≡
    subroutine sf_chain_instance_evaluate (chain, scale)
      class(sf_chain_instance_t), intent(inout) :: chain
      real(default), intent(in) :: scale
      integer :: i, c
      if (chain%status >= SF_DONE_KINEMATICS) then
        if (allocated (chain%sf)) then
          if (size (chain%sf) /= 0) then
            do i = 1, size (chain%sf)
              associate (sf => chain%sf(i))
                call sf%int%apply (scale)
                if (sf%int%status <= SF_FAILED_EVALUATION) then
                  chain%status = SF_FAILED_EVALUATION
                  return
                end if
                if (.not. evaluator_is_empty (sf%eval)) then
                  call evaluator_evaluate (sf%eval)
                end if
              end associate
            end do
          end if
        end if
        chain%status = SF_EVALUATED
      end if
    end subroutine sf_chain_instance_evaluate

```

### 10.3.10 Access to the chain instance

Transfer the outgoing momenta to the array p. We assume that array sizes match.

```

  (SF base: sf chain instance: TBP)+≡
    procedure :: get_out_momenta => sf_chain_instance_get_out_momenta
  (SF base: procedures)+≡
    subroutine sf_chain_instance_get_out_momenta (chain, p)
      class(sf_chain_instance_t), intent(in), target :: chain
      type(vector4_t), dimension(:), intent(out) :: p
      type(interaction_t), pointer :: int
      integer :: i, j
      if (chain%status >= SF_DONE_KINEMATICS) then
        do j = 1, size (chain%out_sf)
          i = chain%out_sf(j)
          select case (i)
            case (0)
              int => beam_get_int_ptr (chain%beam_t)
            case default
              int => chain%sf(i)%int%interaction_t
          end select
          p(j) = interaction_get_momentum (int, chain%out_sf_i(j))
        end do
      end if
    end subroutine sf_chain_instance_get_out_momenta

```

```
end subroutine sf_chain_instance_get_out_momenta
```

Return a pointer to the last evaluator in the chain (to the interaction).

```
<SF base: sf chain instance: TBP>+≡
  procedure :: get_out_int_ptr => sf_chain_instance_get_out_int_ptr

<SF base: procedures>+≡
  function sf_chain_instance_get_out_int_ptr (chain) result (int)
    class(sf_chain_instance_t), intent(in), target :: chain
    type(interaction_t), pointer :: int
    if (chain%out_eval == 0) then
      int => beam_get_int_ptr (chain%beam_t)
    else
      int => evaluator_get_int_ptr (chain%sf(chain%out_eval)%eval)
    end if
  end function sf_chain_instance_get_out_int_ptr
```

Return the index of the j-th outgoing particle, within the last evaluator.

```
<SF base: sf chain instance: TBP>+≡
  procedure :: get_out_i => sf_chain_instance_get_out_i

<SF base: procedures>+≡
  function sf_chain_instance_get_out_i (chain, j) result (i)
    class(sf_chain_instance_t), intent(in) :: chain
    integer, intent(in) :: j
    integer :: i
    i = chain%out_eval_i(j)
  end function sf_chain_instance_get_out_i
```

Return the mask for the outgoing particle(s), within the last evaluator.

```
<SF base: sf chain instance: TBP>+≡
  procedure :: get_out_mask => sf_chain_instance_get_out_mask

<SF base: procedures>+≡
  function sf_chain_instance_get_out_mask (chain) result (mask)
    class(sf_chain_instance_t), intent(in), target :: chain
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
    type(interaction_t), pointer :: int
    allocate (mask (chain%config%n_in))
    int => chain%get_out_int_ptr ()
    mask = interaction_get_mask (int, chain%out_eval_i)
  end function sf_chain_instance_get_out_mask
```

Return the array of MC input parameters that corresponds to channel c. This is the p array, the parameters before all mappings.

The p array may be deallocated. This should correspond to a zero-size r argument, so nothing to do then.

```
<SF base: sf chain instance: TBP>+≡
  procedure :: get_mcpair => sf_chain_instance_get_mcpair
```

```

<SF base: procedures>+=
  subroutine sf_chain_instance_get_mcpair (chain, c, r)
    class(sf_chain_instance_t), intent(in) :: chain
    integer, intent(in) :: c
    real(default), dimension(:), intent(out) :: r
    if (allocated (chain%p)) r = chain%p(:,c)
  end subroutine sf_chain_instance_get_mcpair

```

Return the Jacobian factor that corresponds to channel c.

```

<SF base: sf chain instance: TBP>+=
  procedure :: get_f => sf_chain_instance_get_f

<SF base: procedures>+=
  function sf_chain_instance_get_f (chain, c) result (f)
    class(sf_chain_instance_t), intent(in) :: chain
    integer, intent(in) :: c
    real(default) :: f
    if (allocated (chain%f)) then
      f = chain%f(c)
    else
      f = 1
    end if
  end function sf_chain_instance_get_f

```

Return the evaluation status.

```

<SF base: sf chain instance: TBP>+=
  procedure :: get_status => sf_chain_instance_get_status

<SF base: procedures>+=
  function sf_chain_instance_get_status (chain) result (status)
    class(sf_chain_instance_t), intent(in) :: chain
    integer :: status
    status = chain%status
  end function sf_chain_instance_get_status

```

### 10.3.11 Unit tests

```

<SF base: public>+=
  public :: sf_base_test

<SF base: tests>=
  subroutine sf_base_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <SF base: execute tests>
  end subroutine sf_base_test

```

### 10.3.12 Test implementation: structure function

This is a template for the actual structure-function implementation which will be defined in separate modules.

## Configuration data

The test structure function uses the `Test` model. It describes a scalar within an arbitrary initial particle, which is given in the initialization. The radiated particle is also a scalar, the same one, but we set its mass artificially to zero.

```
<SF base: public>+≡
    public :: sf_test_data_t

<SF base: test types>≡
    type, extends (sf_data_t) :: sf_test_data_t
        type(model_t), pointer :: model => null ()
        integer :: mode = 0
        type(flavor_t) :: flv_in
        type(flavor_t) :: flv_out
        type(flavor_t) :: flv_rad
        real(default) :: m = 0
        logical :: collinear = .true.
        real(default), dimension(:), allocatable :: qbounds
    contains
        <SF base: sf test data: TBP>
    end type sf_test_data_t
```

Output.

```
<SF base: sf test data: TBP>≡
    procedure :: write => sf_test_data_write

<SF base: procedures>+≡
    subroutine sf_test_data_write (data, unit, verbose)
        class(sf_test_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "SF test data:"
        write (u, "(3x,A,A)") "model      = ", char (model_get_name (data%model))
        write (u, "(3x,A)", advance="no") "incoming = "
        call flavor_write (data%flv_in, u); write (u, *)
        write (u, "(3x,A)", advance="no") "outgoing  = "
        call flavor_write (data%flv_out, u); write (u, *)
        write (u, "(3x,A)", advance="no") "radiated  = "
        call flavor_write (data%flv_rad, u); write (u, *)
        write (u, "(3x,A,ES19.12)") "mass      = ", data%m
        write (u, "(3x,A,L1)") "collinear = ", data%collinear
        if (.not. data%collinear .and. allocated (data%qbounds)) then
            write (u, "(3x,A,ES19.12)") "qmin      = ", data%qbounds(1)
            write (u, "(3x,A,ES19.12)") "qmax      = ", data%qbounds(2)
        end if
    end subroutine sf_test_data_write
```

Initialization.

```
<SF base: sf test data: TBP>+≡
    procedure :: init => sf_test_data_init
```

```

<SF base: procedures>+=
subroutine sf_test_data_init (data, model, flv, collinear, qbounds, mode)
  class(sf_test_data_t), intent(out) :: data
  type(model_t), intent(in), target :: model
  type(flavor_t), intent(in) :: flv
  logical, intent(in), optional :: collinear
  real(default), dimension(2), intent(in), optional :: qbounds
  integer, intent(in), optional :: mode
  data%model => model
  if (present (mode)) data%mode = mode
  data%flv_in = flv
  data%m = flavor_get_mass (flv)
  if (present (collinear)) data%collinear = collinear
  call flavor_init (data%flv_out, 25, model)
  call flavor_init (data%flv_rad, 25, model)
  if (present (qbounds)) then
    allocate (data%qbounds (2))
    data%qbounds = qbounds
  end if
end subroutine sf_test_data_init

```

Return the number of parameters: 1 if only consider collinear splitting, 3 otherwise.

```

<SF base: sf test data: TBP>+=
  procedure :: get_n_par => sf_test_data_get_n_par

<SF base: procedures>+=
function sf_test_data_get_n_par (data) result (n)
  class(sf_test_data_t), intent(in) :: data
  integer :: n
  if (data%collinear) then
    n = 1
  else
    n = 3
  end if
end function sf_test_data_get_n_par

```

Return the outgoing particle PDG code: 25

```

<SF base: sf test data: TBP>+=
  procedure :: get_pdg_out => sf_test_data_get_pdg_out

<SF base: procedures>+=
function sf_test_data_get_pdg_out (data) result (pdg_out)
  class(sf_test_data_t), intent(in) :: data
  integer, dimension(:), allocatable :: pdg_out
  allocate (pdg_out (1), source = 25)
end function sf_test_data_get_pdg_out

```

Allocate the matching interaction.

```

<SF base: sf test data: TBP>+=
  procedure :: allocate_sf_int => sf_test_data_allocate_sf_int

```

```

<SF base: procedures>+=
  subroutine sf_test_data_allocate_sf_int (data, sf_int)
    class(sf_test_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (sf_test_t :: sf_int)
  end subroutine sf_test_data_allocate_sf_int

```

## Interaction

```

<SF base: test types>+=
  type, extends (sf_int_t) :: sf_test_t
    type(sf_test_data_t), pointer :: data => null ()
    real(default) :: x = 0
  contains
    <SF base: sf test int: TBP>
  end type sf_test_t

```

Type string: constant

```

<SF base: sf test int: TBP>=
  procedure :: type_string => sf_test_type_string

```

```

<SF base: procedures>+=
  function sf_test_type_string (object) result (string)
    class(sf_test_t), intent(in) :: object
    type(string_t) :: string
    string = "Test"
  end function sf_test_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF base: sf test int: TBP>+=
  procedure :: write => sf_test_write

<SF base: procedures>+=
  subroutine sf_test_write (object, unit)
    class(sf_test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      call object%base_write (u)
    else
      write (u, "(1x,A)") "SF test data: [undefined]"
    end if
  end subroutine sf_test_write

```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

```

(SF base: sf test int: TBP)+≡
  procedure :: init => sf_test_init

(SF base: procedures)+≡
  subroutine sf_test_init (sf_int, data)
    class(sf_test_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    type(helicity_t) :: hel0
    type(quantum_numbers_t), dimension(3) :: qn
    mask = new_quantum_numbers_mask (.false., .false., .false.)
    select type (data)
    type is (sf_test_data_t)
      if (allocated (data%qbounds)) then
        call sf_int%base_init (mask, &
          [data%m**2], [0._default], [data%m**2], &
          [data%qbounds(1)], [data%qbounds(2)])
      else
        call sf_int%base_init (mask, &
          [data%m**2], [0._default], [data%m**2])
      end if
      sf_int%data => data
      call helicity_init (hel0, 0)
      call quantum_numbers_init (qn(1), data%flv_in, hel0)
      call quantum_numbers_init (qn(2), data%flv_rad, hel0)
      call quantum_numbers_init (qn(3), data%flv_out, hel0)
      call interaction_add_state (sf_int%interaction_t, qn)
      call interaction_freeze (sf_int%interaction_t)
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
    end select
    sf_int%status = SF_INITIAL
  end subroutine sf_test_init

```

Set kinematics. If map is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If map is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

(SF base: sf test int: TBP)+≡
  procedure :: complete_kinematics => sf_test_complete_kinematics

(SF base: procedures)+≡
  subroutine sf_test_complete_kinematics (sf_int, x, f, r, map)
    class(sf_test_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    logical, intent(in) :: map
    real(default) :: xb1
    if (map) then
      x(1) = r(1)**2
    end if
  end subroutine sf_test_complete_kinematics

```



```

        f = 2 * r(1)
    else
        x(1) = r(1)
        f = 1
    end if
    xb1 = 1 - x(1)
    if (size (x) == 3)  x(2:3) = r(2:3)
    call sf_int%split_momentum (x, xb1)
    sf_int%x = x(1)
    select case (sf_int%status)
    case (SF_FAILED_KINEMATICS);  f = 0
    end select
end subroutine sf_test_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

⟨SF base: sf test int: TBP⟩+≡
    procedure :: inverse_kinematics => sf_test_inverse_kinematics

⟨SF base: procedures⟩+≡
    subroutine sf_test_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        class(sf_test_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
        real(default) :: xb1
        logical :: set_mom
        set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
        if (map) then
            r(1) = sqrt (x(1))
            f = 2 * r(1)
        else
            r(1) = x(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        if (size (x) == 3)  r(2:3) = x(2:3)
        sf_int%x = x(1)
        if (set_mom) then
            call sf_int%split_momentum (x, xb1)
            select case (sf_int%status)
            case (SF_FAILED_KINEMATICS);  f = 0
            end select
        end if
    end subroutine sf_test_inverse_kinematics

```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

If the mode indicator is one, the matrix element is equal to the parameter  $x$ .

```

⟨SF base: sf test int: TBP⟩+≡

```

```

    procedure :: apply => sf_test_apply
<SF base: procedures>+≡
    subroutine sf_test_apply (sf_int, scale)
        class(sf_test_t), intent(inout) :: sf_int
        real(default), intent(in) :: scale
        select case (sf_int%data%mode)
        case (0)
            call interaction_set_matrix_element (sf_int%interaction_t, &
                cmplx (1._default, kind=default))
        case (1)
            call interaction_set_matrix_element (sf_int%interaction_t, &
                cmplx (sf_int%x, kind=default))
        end select
        sf_int%status = SF_EVALUATED
    end subroutine sf_test_apply

```

### 10.3.13 Test implementation: pair spectrum

Another template, this time for a incoming particle pair, splitting into two radiated and two outgoing particles.

#### Configuration data

For simplicity, the spectrum contains two mirror images of the previous structure-function configuration: the incoming and all outgoing particles are test scalars.

We have two versions, one with radiated particles, one without.

```

<SF base: test types>+≡
    type, extends (sf_data_t) :: sf_test_spectrum_data_t
        type(model_t), pointer :: model => null ()
        type(flavor_t) :: flv_in
        type(flavor_t) :: flv_out
        type(flavor_t) :: flv_rad
        logical :: with_radiation = .true.
        real(default) :: m = 0
    contains
        <SF base: sf test spectrum data: TBP>
    end type sf_test_spectrum_data_t

```

Output.

```

<SF base: sf test spectrum data: TBP>≡
    procedure :: write => sf_test_spectrum_data_write
<SF base: procedures>+≡
    subroutine sf_test_spectrum_data_write (data, unit, verbose)
        class(sf_test_spectrum_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "SF test spectrum data:"
        write (u, "(3x,A,A)") "model      = ", char (model_get_name (data%model))
    end subroutine sf_test_spectrum_data_write

```

```

write (u, "(3x,A)", advance="no") "incoming = "
call flavor_write (data%flv_in, u); write (u, *)
write (u, "(3x,A)", advance="no") "outgoing = "
call flavor_write (data%flv_out, u); write (u, *)
write (u, "(3x,A)", advance="no") "radiated = "
call flavor_write (data%flv_rad, u); write (u, *)
write (u, "(3x,A,ES19.12)") "mass = ", data%m
end subroutine sf_test_spectrum_data_write

```

Initialization.

```

<SF base: sf test spectrum data: TBP>+≡
  procedure :: init => sf_test_spectrum_data_init

<SF base: procedures>+≡
  subroutine sf_test_spectrum_data_init (data, model, flv, with_radiation)
    class(sf_test_spectrum_data_t), intent(out) :: data
    type(model_t), intent(in), target :: model
    type(flavor_t), intent(in) :: flv
    logical, intent(in) :: with_radiation
    data%model => model
    data%with_radiation = with_radiation
    data%flv_in = flv
    data%m = flavor_get_mass (flv)
    call flavor_init (data%flv_out, 25, model)
    if (with_radiation) then
      call flavor_init (data%flv_rad, 25, model)
    end if
  end subroutine sf_test_spectrum_data_init

```

Return the number of parameters: 2, since we have only collinear splitting here.

```

<SF base: sf test spectrum data: TBP>+≡
  procedure :: get_n_par => sf_test_spectrum_data_get_n_par

<SF base: procedures>+≡
  function sf_test_spectrum_data_get_n_par (data) result (n)
    class(sf_test_spectrum_data_t), intent(in) :: data
    integer :: n
    n = 2
  end function sf_test_spectrum_data_get_n_par

```

Return the outgoing particle PDG codes: 25

```

<SF base: sf test spectrum data: TBP>+≡
  procedure :: get_pdg_out => sf_test_spectrum_data_get_pdg_out

<SF base: procedures>+≡
  function sf_test_spectrum_data_get_pdg_out (data) result (pdg_out)
    class(sf_test_spectrum_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    allocate (pdg_out (2), source = 25)
  end function sf_test_spectrum_data_get_pdg_out

```

Allocate the matching interaction.

```

<SF base: sf test spectrum data: TBP>+≡
  procedure :: allocate_sf_int => &
    sf_test_spectrum_data_allocate_sf_int
<SF base: procedures>+≡
  subroutine sf_test_spectrum_data_allocate_sf_int (data, sf_int)
    class(sf_test_spectrum_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (sf_test_spectrum_t :: sf_int)
  end subroutine sf_test_spectrum_data_allocate_sf_int

```

## Interaction

```

<SF base: test types>+≡
  type, extends (sf_int_t) :: sf_test_spectrum_t
    type(sf_test_spectrum_data_t), pointer :: data => null ()
    contains
    <SF base: sf test spectrum: TBP>
  end type sf_test_spectrum_t

<SF base: sf test spectrum: TBP>≡
  procedure :: type_string => sf_test_spectrum_type_string

<SF base: procedures>+≡
  function sf_test_spectrum_type_string (object) result (string)
    class(sf_test_spectrum_t), intent(in) :: object
    type(string_t) :: string
    string = "Test Spectrum"
  end function sf_test_spectrum_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF base: sf test spectrum: TBP>+≡
  procedure :: write => sf_test_spectrum_write

<SF base: procedures>+≡
  subroutine sf_test_spectrum_write (object, unit)
    class(sf_test_spectrum_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      call object%base_write (u)
    else
      write (u, "(1x,A)") "SF test spectrum data: [undefined]"
    end if
  end subroutine sf_test_spectrum_write

```

Initialize. We know that data will be of concrete type `sf_test_spectrum_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

```

(SF base: sf test spectrum: TBP)+≡
  procedure :: init => sf_test_spectrum_init

(SF base: procedures)+≡
  subroutine sf_test_spectrum_init (sf_int, data)
    class(sf_test_spectrum_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(6) :: mask
    type(helicity_t) :: hel0
    type(quantum_numbers_t), dimension(6) :: qn
    mask = new_quantum_numbers_mask (.false., .false., .false.)
    select type (data)
    type is (sf_test_spectrum_data_t)
      if (data%with_radiation) then
        call sf_int%base_init (mask(1:6), &
          [data%m**2, data%m**2], &
          [0._default, 0._default], &
          [data%m**2, data%m**2])
        sf_int%data => data
        call helicity_init (hel0, 0)
        call quantum_numbers_init (qn(1), data%flv_in, hel0)
        call quantum_numbers_init (qn(2), data%flv_in, hel0)
        call quantum_numbers_init (qn(3), data%flv_rad, hel0)
        call quantum_numbers_init (qn(4), data%flv_rad, hel0)
        call quantum_numbers_init (qn(5), data%flv_out, hel0)
        call quantum_numbers_init (qn(6), data%flv_out, hel0)
        call interaction_add_state (sf_int%interaction_t, qn(1:6))
        call sf_int%set_incoming ([1,2])
        call sf_int%set_radiated ([3,4])
        call sf_int%set_outgoing ([5,6])
      else
        call sf_int%base_init (mask(1:4), &
          [data%m**2, data%m**2], &
          [real(default) :: ], &
          [data%m**2, data%m**2])
        sf_int%data => data
        call helicity_init (hel0, 0)
        call quantum_numbers_init (qn(1), data%flv_in, hel0)
        call quantum_numbers_init (qn(2), data%flv_in, hel0)
        call quantum_numbers_init (qn(3), data%flv_out, hel0)
        call quantum_numbers_init (qn(4), data%flv_out, hel0)
        call interaction_add_state (sf_int%interaction_t, qn(1:4))
        call sf_int%set_incoming ([1,2])
        call sf_int%set_outgoing ([3,4])
      end if
      call interaction_freeze (sf_int%interaction_t)
    end select
    sf_int%status = SF_INITIAL
  end subroutine sf_test_spectrum_init

```

Set kinematics. If map is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  (as above) for both  $x$  parameters and consequently  $f(r) = 4r_1r_2$ .

```

(SF base: sf test spectrum: TBP)+≡
  procedure :: complete_kinematics => sf_test_spectrum_complete_kinematics

(SF base: procedures)+≡
  subroutine sf_test_spectrum_complete_kinematics (sf_int, x, f, r, map)
    class(sf_test_spectrum_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    logical, intent(in) :: map
    real(default), dimension(2) :: xb1
    if (map) then
      x = r**2
      f = 4 * r(1) * r(2)
    else
      x = r
      f = 1
    end if
    if (sf_int%data%with_radiation) then
      xb1 = 1 - x
      call sf_int%split_momenta (x, xb1)
    else
      call sf_int%reduce_momenta (x)
    end if
    select case (sf_int%status)
      case (SF_FAILED_KINEMATICS); f = 0
    end select
  end subroutine sf_test_spectrum_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

(SF base: sf test spectrum: TBP)+≡
  procedure :: inverse_kinematics => sf_test_spectrum_inverse_kinematics

(SF base: procedures)+≡
  subroutine sf_test_spectrum_inverse_kinematics &
    (sf_int, x, f, r, map, set_momenta)
    class(sf_test_spectrum_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default), dimension(2) :: xb1
    logical :: set_mom
    set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
    if (map) then
      r = sqrt (x)
      f = 4 * r(1) * r(2)
    else
      r = x
    end if
  end subroutine sf_test_spectrum_inverse_kinematics

```

```

        f = 1
    end if
    if (set_mom) then
        if (sf_int%data%with_radiation) then
            xb1 = 1 - x
            call sf_int%split_momenta (x, xb1)
        else
            call sf_int%reduce_momenta (x)
        end if
        select case (sf_int%status)
        case (SF_FAILED_KINEMATICS); f = 0
        end select
    end if
end subroutine sf_test_spectrum_inverse_kinematics

```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

```

<SF base: sf test spectrum: TBP>+≡
    procedure :: apply => sf_test_spectrum_apply

<SF base: procedures>+≡
    subroutine sf_test_spectrum_apply (sf_int, scale)
        class(sf_test_spectrum_t), intent(inout) :: sf_int
        real(default), intent(in) :: scale
        call interaction_set_matrix_element (sf_int%interaction_t, &
            cmplx (1._default, kind=default))
        sf_int%status = SF_EVALUATED
    end subroutine sf_test_spectrum_apply

```

## Test structure function data

Construct and display a test structure function data object.

```

<SF base: execute tests>≡
    call test (sf_base_1, "sf_base_1", &
        "structure function configuration", &
        u, results)

<SF base: tests>+≡
    subroutine sf_base_1 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable :: data

        write (u, "(A)")  "* Test output: sf_base_1"
        write (u, "(A)")  "* Purpose: initialize and display &
            &test structure function data"
        write (u, "(A)")

        call os_data_init (os_data)
        call syntax_model_file_init ()
        call model_list_read_model (var_str ("Test"), &

```

```

        var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)") "Outgoing particle code:"
write (u, "(2x,99(1x,I0))" data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_1"

end subroutine sf_base_1

```

## Test and probe structure function

Construct and display a structure function object based on the test structure function.

```

<SF base: execute tests>+≡
    call test (sf_base_2, "sf_base_2", &
        "structure function instance", &
        u, results)

<SF base: tests>+≡
subroutine sf_base_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, x
    real(default) :: f, s

    write (u, "(A)")  "* Test output: sf_base_2"
    write (u, "(A)")  "* Purpose: initialize and fill &
        &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

```



```

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

call reset_interaction_counter ()

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=1"
write (u, "(A)")

r = 1
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

```

```

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

r = 0.5_default
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics with mapping for r=0.8"
write (u, "(A)")

r = 0.8_default
call sf_int%complete_kinematics (x, f, r, map=.true.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F12.9))") "x =", x

write (u, "(A)")
write (u, "(A)")  "* Compute inverse kinematics for x=0.64 and evaluate"
write (u, "(A)")

x = 0.64_default
call sf_int%inverse_kinematics (x, f, r, map=.true.)
call sf_int%apply (scale=0._default)

```

```

call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "r =", r
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)") "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)") "* Test output end: sf_base_2"

end subroutine sf_base_2

```

### Collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, collinear case.

```

<SF base: execute tests>+≡
call test (sf_base_3, "sf_base_3", &
  "alternatives for collinear kinematics", &
  u, results)

<SF base: tests>+≡
subroutine sf_base_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  real(default) :: E
  real(default), dimension(:), allocatable :: r, x
  real(default) :: f, s

  write (u, "(A)") "* Test output: sf_base_3"
  write (u, "(A)") "* Purpose: check various kinematical setups"
  write (u, "(A)") "* for collinear structure-function splitting."
  write (u, "(A)") " (two masses equal, one zero)"
  write (u, "(A)")

  write (u, "(A)") "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

```

```

call reset_interaction_counter ()

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%write (u)

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"

E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

```

```

r = 0.5_default
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)

```

```

write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0

```

```

sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_3"

end subroutine sf_base_3

```

### Non-collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, non-collinear case.

```

<SF base: execute tests>+≡
  call test (sf_base_4, "sf_base_4", &
    "alternatives for non-collinear kinematics", &
    u, results)

<SF base: tests>+≡
  subroutine sf_base_4 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    real(default) :: E
    real(default), dimension(:), allocatable :: r, x
    real(default) :: f, s

    write (u, "(A)")  "* Test output: sf_base_4"
    write (u, "(A)")  "*   Purpose: check various kinematical setups"
    write (u, "(A)")  "*               for free structure-function splitting."
    write (u, "(A)")  "               (two masses equal, one zero)"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)
    call flavor_init (flv, 25, model)

    call reset_interaction_counter ()

    allocate (sf_test_data_t :: data)
    select type (data)
    type is (sf_test_data_t)
      call data%init (model, flv, collinear=.false.)
    end select

    write (u, "(A)")  "* Initialize structure-function object"
    write (u, "(A)")

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)

    call sf_int%write (u)

    allocate (r (data%get_n_par ()))
    allocate (x (size (r)))

    write (u, "(A)")
    write (u, "(A)")  "* Initialize incoming momentum with E=500"

```



```

E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"

```

```

write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")

```

```

write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Re-Initialize structure-function object with Q bounds"

call reset_interaction_counter ()

select type (data)
type is (sf_test_data_t)
    call data%init (model, flv, collinear=.false., &
        qbounds = [1._default, 100._default])
end select

call sf_int%init (data)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.false.)

```

```

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.false.)
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_4"

end subroutine sf_base_4

```

## Pair spectrum

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

```

<SF base: execute tests>+≡
  call test (sf_base_5, "sf_base_5", &
    "pair spectrum with radiation", &
    u, results)

<SF base: tests>+≡
  subroutine sf_base_5 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data

```

```

type(model_t), pointer :: model
type(flavor_t) :: flv
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t), dimension(2) :: k
type(vector4_t), dimension(4) :: q
real(default) :: E
real(default), dimension(:), allocatable :: r, x
real(default) :: f, s

write (u, "(A)")  "* Test output: sf_base_5"
write (u, "(A)")  "* Purpose: initialize and fill &
    &a pair spectrum object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

call reset_interaction_counter ()

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
    call data%init (model, flv, with_radiation=.true.)
end select

write (u, "(1x,A)")  "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))")  data%get_pdg_out ()

write (u, "(A)")
write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.4,0.8"

```

```

write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics with mapping for r=0.6,0.8"
write (u, "(A)")

r = [0.6_default, 0.8_default]
call sf_int%complete_kinematics (x, f, r, map=.true.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "x =", x
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call reset_interaction_counter ()
call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%seed_kinematics (k)
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
write (u, "(A,9(1x,F12.9))") "x =", x

write (u, "(A)")
write (u, "(A)")  "* Compute inverse kinematics for x=0.36,0.64 &
    &and evaluate"
write (u, "(A)")

x = [0.36_default, 0.64_default]
call sf_int%inverse_kinematics (x, f, r, map=.true.)
call sf_int%apply (scale=0._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "r =", r

```

```

write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_5"

end subroutine sf_base_5

```

## Pair spectrum without radiation

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

```

<SF base: execute tests>+≡
call test (sf_base_6, "sf_base_6", &
  "pair spectrum without radiation", &
  u, results)

<SF base: tests>+≡
subroutine sf_base_6 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t), dimension(2) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, x
  real(default) :: f, s

  write (u, "(A)")  "* Test output: sf_base_6"
  write (u, "(A)")  "* Purpose: initialize and fill &
    &a pair spectrum object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

  call reset_interaction_counter ()

  allocate (sf_test_spectrum_data_t :: data)

```

```

select type (data)
type is (sf_test_spectrum_data_t)
    call data%init (model, flv, with_radiation=.false.)
end select

write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.4,0.8"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call reset_interaction_counter ()
call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%seed_kinematics (k)
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
write (u, "(A,9(1x,F12.9))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Compute inverse kinematics for x=0.4,0.8 &
    &and evaluate"
write (u, "(A)")

```



```

x = [0.4_default, 0.8_default]
call sf_int%inverse_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale=0._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))") "r =", r
write (u, "(A,9(1x,F12.9))") "f =", f

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_6"

end subroutine sf_base_6

```

## Direct access to structure function

Probe a structure function directly.

```

<SF base: execute tests>+≡
  call test (sf_base_7, "sf_base_7", &
    "direct access", &
    u, results)

<SF base: tests>+≡
subroutine sf_base_7 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  real(default), dimension(:), allocatable :: value

  write (u, "(A)")  "* Test output: sf_base_7"
  write (u, "(A)")  "* Purpose: check direct access method"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

  call reset_interaction_counter ()

```

```

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Probe structure function: states"
write (u, "(A)")

write (u, "(A,IO)")  "n_states = ", sf_int%get_n_states ()
write (u, "(A,IO)")  "n_in      = ", sf_int%get_n_in ()
write (u, "(A,IO)")  "n_rad     = ", sf_int%get_n_rad ()
write (u, "(A,IO)")  "n_out     = ", sf_int%get_n_out ()
write (u, "(A)")
write (u, "(A)", advance="no")  "state(1) = "
call quantum_numbers_write (sf_int%get_state (1), u)
write (u, *)

allocate (value (sf_int%get_n_states ()))
call sf_int%compute_values (value, &
    E=[500._default], x=[0.5_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.5) ="
write (u, "(9(1x,ES19.12))")  value

call sf_int%compute_values (value, &
    x=[0.1_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.1) ="
write (u, "(9(1x,ES19.12))")  value

write (u, "(A)")
write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

deallocate (value)
call sf_int%final ()
deallocate (sf_int)
deallocate (data)

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
    call data%init (model, flv, with_radiation=.false.)

```

```

end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Probe spectrum: states"
write (u, "(A)")

write (u, "(A,IO)") "n_states = ", sf_int%get_n_states ()
write (u, "(A,IO)") "n_in      = ", sf_int%get_n_in ()
write (u, "(A,IO)") "n_rad     = ", sf_int%get_n_rad ()
write (u, "(A,IO)") "n_out     = ", sf_int%get_n_out ()
write (u, "(A)")
write (u, "(A)", advance="no") "state(1) = "
call quantum_numbers_write (sf_int%get_state (1), u)
write (u, *)

allocate (value (sf_int%get_n_states ()))
call sf_int%compute_value (1, value(1), &
    E=[500._default, 500._default], x=[0.5_default, 0.6_default], &
    scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no") "value (E=500,500, x=0.5,0.6) ="
write (u, "(9(1x,ES19.12))") value

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_7"

end subroutine sf_base_7

```

## Structure function chain configuration

```

<SF base: execute tests>+≡
    call test (sf_base_8, "sf_base_8", &
        "structure function chain configuration", &
        u, results)

<SF base: tests>+≡
subroutine sf_base_8 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(beam_data_t), target :: beam_data
    class(sf_data_t), allocatable, target :: data_strfun
    class(sf_data_t), allocatable, target :: data_spectrum

```

```

type(sf_config_t), dimension(:), allocatable :: sf_config
type(sf_chain_t) :: sf_chain

write (u, "(A)")  "* Test output: sf_base_8"
write (u, "(A)")  "* Purpose: set up a structure-function chain"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

call reset_interaction_counter ()

call beam_data_init_sqrt (beam_data, &
    1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
    call data_strfun%init (model, flv)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
    call data_spectrum%init (model, flv, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)
call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain%final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_8"

end subroutine sf_base_8

```

## Structure function instance configuration

We create a structure-function chain instance which implements a configured structure-function chain. We link the momentum entries in the interactions and compute kinematics.

We do not actually connect the interactions and create evaluators. We skip this step and manually advance the status of the chain instead.

```

<SF base: execute tests>+≡
  call test (sf_base_9, "sf_base_9", &
    "structure function chain instance", &
    u, results)

<SF base: tests>+≡
  subroutine sf_base_9 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(beam_data_t), target :: beam_data
    class(sf_data_t), allocatable, target :: data_strfun
    class(sf_data_t), allocatable, target :: data_spectrum
    type(sf_config_t), dimension(:), allocatable, target :: sf_config
    type(sf_chain_t), target :: sf_chain
    type(sf_chain_instance_t), target :: sf_chain_instance
    type(sf_channel_t), dimension(2) :: sf_channel
    type(vector4_t), dimension(2) :: p
    integer :: j

```

```

write (u, "(A)")  "* Test output: sf_base_9"
write (u, "(A)")  "* Purpose: set up a structure-function chain &
                    &and create an instance"
write (u, "(A)")  "* compute kinematics"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

call reset_interaction_counter ()

call beam_data_init_sqrt (beam_data, &
    1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
    call data_strfun%init (model, flv)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
    call data_spectrum%init (model, flv, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [real(default) ::])

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

```

```

do j = 1, 2
    write (u, "(A)")
    call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "** Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [0.8_default])

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "** Outgoing momenta:"

do j = 1, 2
    write (u, "(A)")
    call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "** Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)

```

```

call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics &
    (1, [0.5_default, 0.6_default, 0.8_default])

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

do j = 1, 2
    write (u, "(A)")
    call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_9"

end subroutine sf_base_9

```

## Structure function chain mappings

Set up a structure function chain instance with a pair of single-particle structure functions. We test different global mappings for this setup.

Again, we skip evaluators.

```

<SF base: execute tests>+≡
call test (sf_base_10, "sf_base_10", &
    "structure function chain mapping", &
    u, results)

```



$\langle SF \text{ base: tests} \rangle + \equiv$

```

subroutine sf_base_10 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  type(beam_data_t), target :: beam_data
  class(sf_data_t), allocatable, target :: data_strfun
  type(sf_config_t), dimension(:), allocatable, target :: sf_config
  type(sf_chain_t), target :: sf_chain
  type(sf_chain_instance_t), target :: sf_chain_instance
  type(sf_channel_t), dimension(2) :: sf_channel
  real(default), dimension(2) :: x_saved

  write (u, "(A)")  "* Test output: sf_base_10"
  write (u, "(A)")  "*   Purpose: set up a structure-function chain"
  write (u, "(A)")  "*                   and check mappings"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

  call reset_interaction_counter ()

  call beam_data_init_sqrts (beam_data, &
    1000._default, [flv, flv])

  allocate (sf_test_data_t :: data_strfun)
  select type (data_strfun)
  type is (sf_test_data_t)
    call data_strfun%init (model, flv)
  end select

  write (u, "(A)")  "* Set up chain with structure function pair &
    &and standard mapping"
  write (u, "(A)")

  allocate (sf_config (2))
  call sf_config(1)%init ([1], data_strfun)
  call sf_config(2)%init ([2], data_strfun)
  call sf_chain%init (beam_data, sf_config)

  call sf_chain_instance%init (sf_chain, n_channel = 1)

  call sf_channel(1)%init (2)
  call sf_channel(1)%set_s_mapping ([1,2])
  call sf_chain_instance%set_channel (1, sf_channel(1))

```

```

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Invert the kinematics calculation"
write (u, "(A)")

x_saved = sf_chain_instance%x

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(2)%init (2)
call sf_channel(2)%set_s_mapping ([1, 2])
call sf_chain_instance%set_channel (1, sf_channel(2))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%inverse_kinematics (x_saved)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%final ()
call sf_chain%final ()

!   write (u, "(A)")
!   write (u, "(A)")  "* Set up chain with structure function pair &
!       &and s-channel resonance mapping"
!   write (u, "(A)")
!
!   call sf_chain%init (beam_data, n_strfun = 2)
!   call sf_chain%set_strfun (1, [1], data_strfun)
!   call sf_chain%set_strfun (2, [2], data_strfun)
!
!   call sf_chain_instance%init (sf_chain, n_channel = 1)
! !   call sf_chain_instance%set_resonance_mapping (1, 2, 200._default, 5._default)
!
!   call sf_chain_instance%link_interactions ()
!   sf_chain_instance%status = SF_DONE_CONNECTIONS
!   call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])
!
!   call write_separator_double (u)
!   call sf_chain_instance%write (u)
!   call write_separator_double (u)
!
!   write (u, "(A)")
!   write (u, "(A)")  "* Invert the kinematics calculation"

```

```

!       write (u, "(A)")
!
!       x_saved = sf_chain_instance%x
!
!       call sf_chain_instance%init (sf_chain, n_channel = 1)
! !       call sf_chain_instance%set_resonance_mapping (1, 2, 200._default, 5._default)
!
!       call sf_chain_instance%link_interactions ()
!       sf_chain_instance%status = SF_DONE_CONNECTIONS
!       call sf_chain_instance%inverse_kinematics (x_saved)
!
!       call write_separator_double (u)
!       call sf_chain_instance%write (u)
!       call write_separator_double (u)
!
!
!       call sf_chain_instance%final ()
!       call sf_chain%final ()
!
!       write (u, "(A)")
!       write (u, "(A)")  "* Set up chain with structure function pair &
!           &and standard mapping"
!       write (u, "(A)")
!
!       call sf_chain%init (beam_data, n_strfun = 2)
!       call sf_chain%set_strfun (1, [1], data_strfun)
!       call sf_chain%set_strfun (2, [2], data_strfun)
!
!       call sf_chain_instance%init (sf_chain, n_channel = 1)
! !       call sf_chain_instance%set_on_shell_mapping (1, 2, 200._default)
!
!       call sf_chain_instance%link_interactions ()
!       sf_chain_instance%status = SF_DONE_CONNECTIONS
!       call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])
!
!       call write_separator_double (u)
!       call sf_chain_instance%write (u)
!       call write_separator_double (u)
!
!       write (u, "(A)")
!       write (u, "(A)")  "* Invert the kinematics calculation"
!       write (u, "(A)")
!
!       x_saved = sf_chain_instance%x
!
!       call sf_chain_instance%init (sf_chain)
! !       call sf_chain_instance%set_on_shell_mapping (1, 2, 200._default)
!
!       call sf_chain_instance%link_interactions ()
!       sf_chain_instance%status = SF_DONE_CONNECTIONS
!       call sf_chain_instance%inverse_kinematics (x_saved)
!
!       call write_separator_double (u)
!       call sf_chain_instance%write (u)

```

```

!      call write_separator_double (u)
!
!
!      call sf_chain_instance%final ()
!      call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_10"

end subroutine sf_base_10

```

## Structure function chain evaluation

Here, we test the complete workflow for structure-function chains. First, we create the template chain, then initialize an instance. We set up links, mask, and evaluators. Finally, we set kinematics and evaluate the matrix elements and their products.

```

<SF base: execute tests>+≡
call test (sf_base_11, "sf_base_11", &
          "structure function chain evaluation", &
          u, results)

<SF base: tests>+≡
subroutine sf_base_11 (u)
integer, intent(in) :: u
type(os_data_t) :: os_data
type(model_t), pointer :: model
type(flavor_t) :: flv
type(beam_data_t), target :: beam_data
class(sf_data_t), allocatable, target :: data_strfun
class(sf_data_t), allocatable, target :: data_spectrum
type(sf_config_t), dimension(:), allocatable, target :: sf_config
type(sf_chain_t), target :: sf_chain
type(sf_chain_instance_t), target :: sf_chain_instance
type(sf_channel_t), dimension(2) :: sf_channel
type(particle_set_t) :: pset
type(interaction_t), pointer :: int
logical :: ok

write (u, "(A)")  "* Test output: sf_base_11"
write (u, "(A)")  "* Purpose: set up a structure-function chain"
write (u, "(A)")  "* create an instance and evaluate"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)

```

```

call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
    1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
    call data_strfun%init (model, flv)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
    call data_spectrum%init (model, flv, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics (1, [real(default) ::])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
    [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"

```

```

write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_fill_interaction (pset, int, 2)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call particle_set_final (pset)
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics (1, [0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
    [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

```

```

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_fill_interaction (pset, int, 2)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call particle_set_final (pset)
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")  "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics &
      (1, [0.5_default, 0.6_default, 0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

```

```

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
    [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_fill_interaction (pset, int, 2)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call particle_set_final (pset)
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_11"

end subroutine sf_base_11

```



## Multichannel case

We set up a structure-function chain as before, but with three different parameterizations. The first instance is without mappings, the second one with single-particle mappings, and the third one with two-particle mappings.

```
<SF base: execute tests>+≡
  call test (sf_base_12, "sf_base_12", &
    "multi-channel structure function chain", &
    u, results)

<SF base: tests>+≡
  subroutine sf_base_12 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(beam_data_t), target :: beam_data
    class(sf_data_t), allocatable, target :: data
    type(sf_config_t), dimension(:), allocatable, target :: sf_config
    type(sf_chain_t), target :: sf_chain
    type(sf_chain_instance_t), target :: sf_chain_instance
    real(default), dimension(2) :: x_saved
    real(default), dimension(2,3) :: p_saved
    type(sf_channel_t), dimension(:), allocatable :: sf_channel

    write (u, "(A)")  "* Test output: sf_base_12"
    write (u, "(A)")  "*   Purpose: set up and evaluate a multi-channel &
      &structure-function chain"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)
    call flavor_init (flv, 25, model)

    call reset_interaction_counter ()

    call beam_data_init_sqrts (beam_data, &
      1000._default, [flv, flv])

    allocate (sf_test_data_t :: data)
    select type (data)
    type is (sf_test_data_t)
      call data%init (model, flv)
    end select

    write (u, "(A)")  "* Set up chain with structure function pair &
      &and three different mappings"
    write (u, "(A)")

    allocate (sf_config (2))
```

```

call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 3)

call allocate_sf_channels (sf_chain, n_channel = 3, n_strfun = 2)

! channel 1: no mapping
call sf_chain_instance%set_channel (1, sf_channel(1))

! channel 2: single-particle mappings
call sf_channel(2)%activate_mapping ([1,2])
! call sf_chain_instance%activate_mapping (2, [1,2])
call sf_chain_instance%set_channel (2, sf_channel(2))

! channel 3: two-particle mapping
call sf_channel(3)%set_s_mapping ([1,2])
! call sf_chain_instance%set_s_mapping (3, [1, 2])
call sf_chain_instance%set_channel (3, sf_channel(3))

call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

write (u, "(A)")  "* Compute kinematics in channel 1 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Invert the kinematics calculation"
write (u, "(A)")

x_saved = sf_chain_instance%x

call sf_chain_instance%inverse_kinematics (x_saved)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 2 and evaluate"
write (u, "(A)")

p_saved = sf_chain_instance%p

```

```

call sf_chain_instance%compute_kinematics (2, p_saved(:,2))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 3 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (3, p_saved(:,3))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_chain_instance%final ()
call sf_chain%final ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_base_12"

end subroutine sf_base_12

```

## 10.4 Photon radiation: ISR

```

(sf_isr.f90)≡
<File header>

module sf_isr

  <Use kinds>
  <Use strings>
  use constants, only: pi !NODEP!
  <Use file utils>
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use sm_physics, only: Li2 !NODEP!
  use unit_tests
  use os_interface
  use models
  use flavors
  use colors
  use quantum_numbers
  use state_matrices

```

```

    use polarizations
    use interactions
    use sf_aux
    use sf_base

    <Standard module head>

    <SF isr: public>

    <SF isr: parameters>

    <SF isr: types>

    contains

    <SF isr: procedures>

    <SF isr: tests>

    end module sf_isr

```

### 10.4.1 Physics

The ISR structure function is in the most crude approximation (LLA without  $\alpha$  corrections, i.e.  $\epsilon^0$ )

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \quad \text{with} \quad \epsilon = \frac{\alpha}{\pi} q_e^2 \ln \frac{s}{m^2}, \quad (10.21)$$

where  $m$  is the mass of the incoming (and outgoing) particle, which is initially assumed on-shell.

Here, the form of  $\epsilon$  results from the kinematical bounds for the momentum squared of the outgoing particle, which in the limit  $m^2 \ll s$  are given by

$$t_0 = -2\bar{x}E(E+p) + m^2 \approx -\bar{x}s, \quad (10.22)$$

$$t_1 = -2\bar{x}E(E-p) + m^2 \approx xm^2, \quad (10.23)$$

so the integration over the propagator  $1/(t-m^2)$  yields

$$\ln \frac{t_0 - m^2}{t_1 - m^2} = \ln \frac{s}{m^2}. \quad (10.24)$$

In  $f_0(x)$ , there is an integrable singularity at  $x = 1$  which does not spoil the integration, but would lead to an unbounded  $f_{\max}$ . Therefore, we map this singularity like

$$x = 1 - (1-x')^{1/\epsilon} \quad (10.25)$$

such that

$$\int dx f_0(x) = \int dx' \quad (10.26)$$

The structure function has three parameters:  $\alpha$ ,  $m_{\text{in}}$  of the incoming particle and  $s$ , the hard scale. Internally, we store the exponent  $\epsilon$  which is the relevant parameter. (In conventional notation,  $\epsilon = \beta/2$ .) As defaults, we take the actual

values of  $\alpha$  (which is probably  $\alpha(s)$ ), the actual mass  $m_{\text{in}}$  and the squared total c.m. energy  $s$ .

Including  $\epsilon$ ,  $\epsilon^2$ , and  $\epsilon^3$  corrections, the successive approximation of the ISR structure function read

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \quad (10.27)$$

$$f_1(x) = g_1(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x) \quad (10.28)$$

$$f_2(x) = g_2(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x) - \frac{\epsilon^2}{8} \left( \frac{1+3x^2}{1-x} \ln x + 4(1+x) \ln(1-x) + 5+x \right) \quad (10.29)$$

$$\begin{aligned} f_3(x) = g_3(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x) & - \frac{\epsilon^2}{8} \left( \frac{1+3x^2}{1-x} \ln x + 4(1+x) \ln(1-x) + 5+x \right) \\ & - \frac{\epsilon^3}{48} \left( (1+x) [6 \text{Li}_2(x) + 12 \ln^2(1-x) - 3\pi^2] + 6(x+5) \ln(1-x) \right. \\ & \quad \left. + \frac{1}{1-x} \left[ \frac{3}{2}(1+8x+3x^2) \ln x + 12(1+x^2) \ln x \ln(1-x) \right. \right. \\ & \quad \left. \left. - \frac{1}{2}(1+7x^2) \ln^2 x + \frac{1}{4}(39-24x-15x^2) \right] \right) \end{aligned} \quad (10.30)$$

where the successive approximations to the prefactor of the leading singularity

$$g(\epsilon) = \frac{\exp(\epsilon(-\gamma_E + \frac{3}{4}))}{\Gamma(1+\epsilon)}, \quad (10.31)$$

are given by

$$g_0(\epsilon) = 1 \quad (10.32)$$

$$g_1(\epsilon) = 1 + \frac{3}{4}\epsilon \quad (10.33)$$

$$g_2(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 \quad (10.34)$$

$$g_3(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 + \frac{27-24\pi^2+128\zeta(3)}{384}\epsilon^3, \quad (10.35)$$

where, numerically

$$\zeta(3) = 1.20205690315959428539973816151 \dots \quad (10.36)$$

Although one could calculate the function  $g(\epsilon)$  exactly, truncating its Taylor expansion ensures the exact normalization of the truncated structure function at each given order:

$$\int_0^1 dx f_i(x) = 1 \quad \text{for all } i. \quad (10.37)$$

Effectively, the  $O(\epsilon)$  correction reduces the low- $x$  tail of the structure function by 50% while increasing the coefficient of the singularity by  $O(\epsilon)$ . Relative

to this, the  $O(\epsilon^2)$  correction slightly enhances  $x > \frac{1}{2}$  compared to  $x < \frac{1}{2}$ . At  $x = 0$ ,  $f_2(x)$  introduces a logarithmic singularity which should be cut off at  $x_0 = O(e^{-1/\epsilon})$ : for lower  $x$  the perturbative series breaks down. The  $f_3$  correction is slightly positive for low  $x$  values and negative near  $x = 1$ , where the  $\text{Li}_2$  piece slightly softens the singularity at  $x = 1$ .

Instead of the definition for  $\epsilon$  given above, it is customary to include a universal nonlogarithmic piece:

$$\epsilon = \frac{\alpha}{\pi} q_e^2 \left( \ln \frac{s}{m^2} - 1 \right) \quad (10.38)$$

### 10.4.2 Implementation

In the concrete implementation, the zeroth order mapping (10.25) is implemented, and the Jacobian is equal to  $f_i(x)/f_0(x)$ . This can be written as

$$\frac{f_0(x)}{f_0(x)} = 1 \quad (10.39)$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon - \frac{1-x^2}{2(1-x')} \quad (10.40)$$

$$\begin{aligned} \frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 - \frac{1-x^2}{2(1-x')} \\ - \frac{(1+3x^2)\ln x + (1-x)(4(1+x)\ln(1-x) + 5+x)}{8(1-x')}\epsilon \end{aligned} \quad (10.41)$$

For  $x = 1$  (i.e., numerically indistinguishable from 1), this reduces to

$$\frac{f_0(x)}{f_0(x)} = 1 \quad (10.42)$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon \quad (10.43)$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 \quad (10.44)$$

The last line in (10.41) is zero for

$$x_{\min} = 0.00714053329734592839549879772019 \quad (10.45)$$

(Mathematica result), independent of  $\epsilon$ . For  $x$  values less than this we ignore this correction because of the logarithmic singularity which should in principle be resummed.

### 10.4.3 The ISR data block

```

<SF isr: public>≡
  public :: isr_data_t

<SF isr: types>≡
  type, extends (sf_data_t) :: isr_data_t
  private
  type(model_t), pointer :: model => null ()
  type(flavor_t) :: flv

```

```

real(default) :: alpha = 0
real(default) :: q_max = 0
real(default) :: real_mass = 0
real(default) :: mass = 0
real(default) :: eps = 0
real(default) :: log = 0
logical :: recoil = .false.
integer :: order = 3
integer :: error = NONE
logical :: map = .true.
contains
<SF isr: isr data: TBP>
end type isr_data_t

```

Error codes

```

<SF isr: parameters>≡
integer, parameter :: NONE = 0
integer, parameter :: ZERO_MASS = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: EPS_TOO_LARGE = 3
integer, parameter :: INVALID_ORDER = 4

```

Generate flavor-dependent ISR data:

```

<SF isr: isr data: TBP>≡
procedure :: init => isr_data_init

<SF isr: procedures>≡
subroutine isr_data_init &
  (data, model, flv, alpha, q_max, mass, order, recoil)
class(isr_data_t), intent(out) :: data
type(model_t), intent(in), target :: model
type(flavor_t), intent(in) :: flv
real(default), intent(in) :: alpha
real(default), intent(in) :: q_max
real(default), intent(in), optional :: mass
integer, intent(in), optional :: order
logical, intent(in), optional :: recoil
data%model => model
data%flv = flv
data%alpha = alpha
data%q_max = q_max
if (present (order)) then
  call data%set_order (order)
end if
if (present (recoil)) then
  data%recoil = recoil
end if
data%real_mass = flavor_get_mass (flv)
if (present (mass)) then
  if (mass > 0) then
    data%mass = mass
  else
    data%mass = data%real_mass
  end if
end if

```

```

else
    data%mass = data%real_mass
end if
if (data%mass == 0) then
    data%error = ZERO_MASS; return
else if (data%mass >= data%q_max) then
    data%error = Q_MAX_TOO_SMALL; return
end if
data%log = log (1 + (data%q_max / data%mass)**2)
data%eps = data%alpha / pi &
    * flavor_get_charge (data%flv)**2 &
    * (2 * log (data%q_max / data%mass) - 1)
if (data%eps > 1) then
    data%error = EPS_TOO_LARGE; return
end if
end subroutine isr_data_init

```

Explicitly set ISR order

```

<SF isr: isr data: TBP>+≡
    procedure :: set_order => isr_data_set_order

<SF isr: procedures>+≡
    elemental subroutine isr_data_set_order (data, order)
        class(isr_data_t), intent(inout) :: data
        integer, intent(in) :: order
        if (order < 0 .or. order > 3) then
            data%error = INVALID_ORDER
        else
            data%order = order
        end if
    end subroutine isr_data_set_order

```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

```

<SF isr: isr data: TBP>+≡
    procedure :: check => isr_data_check

<SF isr: procedures>+≡
    subroutine isr_data_check (data)
        class(isr_data_t), intent(in) :: data
        select case (data%error)
        case (ZERO_MASS)
            call msg_fatal (" ISR: Particle mass is zero")
        case (Q_MAX_TOO_SMALL)
            call msg_fatal (" ISR: Particle mass exceeds Qmax")
        case (EPS_TOO_LARGE)
            call msg_fatal (" ISR: Expansion parameter too large, " // &
                "perturbative expansion breaks down")
        case (INVALID_ORDER)
            call msg_error (" ISR: LLA order invalid (valid values are 0,1,2,3)")
        end select
    end subroutine isr_data_check

```



Output

```

<SF isr: isr data: TBP>+≡
  procedure :: write => isr_data_write

<SF isr: procedures>+≡
  subroutine isr_data_write (data, unit, verbose) !, md5)
    class(isr_data_t), intent(in) :: data
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    !!! JRR: WK please check: still needed?
    ! logical, intent(in), optional :: md5
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A)") "ISR data:"
    if (flavor_is_defined (data%flv)) then
      write (u, "(3x,A)", advance="no") " flavor = "
      call flavor_write (data%flv, u); write (u, *)
      write (u, "(3x,A,ES19.12)") " alpha = ", data%alpha
      write (u, "(3x,A,ES19.12)") " q_max = ", data%q_max
      write (u, "(3x,A,ES19.12)") " mass = ", data%mass
      write (u, "(3x,A,ES19.12)") " eps = ", data%eps
      write (u, "(3x,A,ES19.12)") " log = ", data%log
      write (u, "(3x,A,I2)") " order = ", data%order
      write (u, "(3x,A,L2)") " recoil = ", data%recoil
      write (u, "(3x,A,L2)") " map = ", data%map
    else
      write (u, "(3x,A)") "[undefined]"
    end if
  end subroutine isr_data_write

```

For ISR, there is the option to generate transverse momentum is generated. Hence, there can be up to three parameters,  $x$ , and two angles.

```

<SF isr: isr data: TBP>+≡
  procedure :: get_n_par => isr_data_get_n_par

<SF isr: procedures>+≡
  function isr_data_get_n_par (data) result (n)
    class(isr_data_t), intent(in) :: data
    integer :: n
    if (data%recoil) then
      n = 3
    else
      n = 1
    end if
  end function isr_data_get_n_par

```

Return the outgoing particles PDG codes.

```

<SF isr: isr data: TBP>+≡
  procedure :: get_pdg_out => isr_data_get_pdg_out

<SF isr: procedures>+≡
  !!! JRR: WK please check
  function isr_data_get_pdg_out (data) result (pdg_out)
    class(isr_data_t), intent(in) :: data

```

```

integer, dimension(:), allocatable :: pdg_out
integer :: i, n
n = 1
allocate (pdg_out (n))
pdg_out(1) = flavor_get_pdg (data%flv)
end function isr_data_get_pdg_out

```

Allocate the interaction record.

```

<SF isr: data: TBP>+≡
  procedure :: allocate_sf_int => isr_data_allocate_sf_int

<SF isr: procedures>+≡
  subroutine isr_data_allocate_sf_int (data, sf_int)
    class(isr_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (isr_t :: sf_int)
  end subroutine isr_data_allocate_sf_int

```

#### 10.4.4 The ISR object

The `isr_t` data type is a  $1 \rightarrow 2$  interaction, i.e., we allow for single-photon emission only (but use the multi-photon resummed radiator function). The particles are ordered as (incoming, photon, outgoing).

There is no need to handle several flavors (and data blocks) in parallel, since ISR is always applied immediately after beam collision. (ISR for partons is accounted for by the PDFs themselves.) Polarization is carried through, i.e., we retain the polarization of the incoming particle and treat the emitted photon as unpolarized. Color is trivially carried through. This implies that particles 1 and 3 should be locked together. !!! JRR: WK please check For ISR we don't seem to need the `q` variable.

```

<SF isr: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: isr_t
    type(isr_data_t), pointer :: data => null ()
    real(default) :: x = 0
  contains
    <SF isr: isr: TBP>
  end type isr_t

```

Type string: has to here, but there is no string variable on which ISR depends. Hence, a dummy routine.

```

<SF isr: isr: TBP>≡
  procedure :: type_string => isr_type_string

<SF isr: procedures>+≡
  function isr_type_string (object) result (string)
    class(isr_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "ISR: e+ e- ISR spectrum"
    else

```

```

        string = "ISR: [undefined]"
    end if
end function isr_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF isr: isr: TBP>+≡
    procedure :: write => isr_write

<SF isr: procedures>+≡
    subroutine isr_write (object, unit)
        !!! JRR: WK please check
        class(isr_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        if (associated (object%data)) then
            call object%data%write (u)
            if (object%status >= SF_DONE_KINEMATICS) then
                write (u, "(1x,A)") "SF parameters:"
                write (u, "(3x,A,ES17.10)") "x =", object%x
            end if
            call object%base_write (u)
        else
            write (u, "(1x,A)") "ISR data: [undefined]"
        end if
    end subroutine isr_write

```

#### 10.4.5 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  were trivial. The ISR structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the  $x$  value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number  $\mathbf{r}$  to generate the momentum fraction  $\mathbf{x}$  and the function value  $\mathbf{f}$ . For numerical stability reasons, we also output  $\mathbf{x_b}$ , which is  $\bar{x} = 1 - x$ . The mapping ensures that  $f = 1$  at leading order, and the higher-order corrections are well-behaved. So, the Jacobian for the kinematics is one if the mapping is applied as it is already defined within the value of the structure function. JRR: WK please check!

If the `no_map` flag is set, we do not apply this mapping.

```

<SF isr: isr: TBP>+≡
    procedure :: complete_kinematics => isr_complete_kinematics

<SF isr: procedures>+≡
    subroutine isr_complete_kinematics (sf_int, x, f, r, map)
        !!! JRR: WK please check
        class(isr_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(out) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: r
        logical, intent(in) :: map
    end subroutine isr_complete_kinematics

```

```

real(default) :: rb1, xb1, eps
eps = sf_int%data%eps
!!!
rb1 = 1 - r(1)
if (map) then
  if (rb1 < tiny(1._default)**eps) then
    xb1 = 0
  else
    xb1 = rb1**(1/eps)
  end if
  f = 1
else
  xb1 = rb1
  f = xb1 ** (-1 + eps)
end if
x(1) = 1 - xb1
if (size (x) == 3) x(2:3) = r(2:3)
call sf_int%split_momentum (x, xb1)
!!! JRR: WK please check
!!! in kinematics we have to set the map flag in order
!!! to be available in apply
sf_int%data%map = map
select case (sf_int%status)
case (SF_DONE_KINEMATICS)
  sf_int%x = x(1)
case (SF_FAILED_KINEMATICS)
  sf_int%x = 0
  f = 0
end select
end subroutine isr_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics. Note that for the ISR SF the standard Jacobian for the mapping is defined into the SF value. (JRR: WK please check)

```

<SF isr: isr: TBP>+≡
  procedure :: inverse_kinematics => isr_inverse_kinematics

<SF isr: procedures>+≡
  subroutine isr_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
    !!! JRR: WK please check
    class(isr_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default) :: xb1, rb1, eps
    logical :: set_mom
    set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
    eps = sf_int%data%eps
    xb1 = 1 - x(1)
    if (map) then
      !!! JRR: WK please check

```

```

    !!! Do I need to introduce a "tiny" statement here, too?
    rb1 = xb1**eps
    f = 1
else
    rb1 = xb1
    f = xb1 ** (-1 + eps)
end if
r(1) = 1 - rb1
if (size(r) == 3) r(2:3) = x(2:3)
!!! JRR: WK please check
!!! We have to set the map flag here
sf_int%data%map = map
if (set_mom) then
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end if
end subroutine isr_inverse_kinematics

```

$\langle SF \text{ isr: isr: TBP} \rangle + \equiv$

```

procedure :: init => isr_init

```

$\langle SF \text{ isr: procedures} \rangle + \equiv$

```

subroutine isr_init (sf_int, data)
    !!! JRR: WK please check
    class(isr_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    integer, dimension(3) :: hel_lock
    type(polarization_t) :: pol
    type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
    type(flavor_t) :: flv_photon
    type(quantum_numbers_t) :: qn_photon, qn
    type(state_iterator_t) :: it_hel
    real(default) :: m2
    mask = new_quantum_numbers_mask (.false., .false., &
        mask_h = (/ .false., .true., .false. /))
    hel_lock = (/ 3, 0, 1 /)
    select type (data)
    type is (isr_data_t)
        m2 = data%mass**2
        call sf_int%base_init (mask, [m2], [0._default], [m2], &
            hel_lock = hel_lock)
        sf_int%data => data
        call flavor_init (flv_photon, PHOTON, data%model)
        call quantum_numbers_init (qn_photon, flv_photon)
        call polarization_init_generic (pol, data%flv)
        call quantum_numbers_init (qn_fc(1), &
            flv = data%flv, col = color_from_flavor (data%flv, 1))
    end select
end subroutine isr_init

```

```

      call state_iterator_init (it_hel, pol%state)
      do while (state_iterator_is_valid (it_hel))
        qn_hel = state_iterator_get_quantum_numbers (it_hel)
        qn = qn_hel(1) .merge. qn_fc(1)
        call interaction_add_state &
          (sf_int%interaction_t, (/ qn, qn_photon, qn /))
        call state_iterator_advance (it_hel)
      end do
      call polarization_final (pol)
      call interaction_freeze (sf_int%interaction_t)
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
      sf_int%status = SF_INITIAL
    end select
  end subroutine isr_init

```

```

<CCC SF isr: procedures>≡
  subroutine isr_strfun (f, x, xb, r, data, no_map)
    real(default), intent(out) :: f, x, xb
    real(default), intent(in) :: r
    type(isr_data_t), intent(in) :: data
    logical, intent(in) :: no_map
    real(default) :: eps
    real(default) :: rb, log_x, log_xb, x_2

    eps = data%eps

  end subroutine isr_strfun

```

#### 10.4.6 ISR application

For ISR, we could in principle compute kinematics and function value in a single step. In order to be able to reweight matrix elements including structure functions we split kinematics and structure function calculation. The structure function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for  $x$ , one for  $Q^2$ , and two for the polar and azimuthal angles. Alternatively, we can skip  $p_T$  generation; in this case, we only need one. JRR: WK please check do we really need four variables? We do we need to generate  $Q^2$ ?

After splitting momenta, we set the outgoing momenta on-shell. We choose to conserve momentum, so energy conservation may be violated.

```

<SF isr: TBP>+≡
  procedure :: apply => isr_apply

<SF isr: procedures>+≡
  subroutine isr_apply (sf_int, scale) !!! r, isr_data, no_map)
    !!! JRR: WK please check
    class(isr_t), intent(inout) :: sf_int
    !!! Scale is only a dummy variable here.
    real(default), intent(in) :: scale

```

```

real(default) :: f, x, xb, eps, r, rb
real(default) :: log_x, log_xb, x_2
real(default), parameter :: &
    & xmin = 0.00714053329734592839549879772019_default
real(default), parameter :: &
    & zeta3 = 1.20205690315959428539973816151_default
real(default), parameter :: &
    & g1 = 3._default / 4._default, &
    & g2 = (27 - 8*pi**2) / 96._default, &
    & g3 = (27 - 24*pi**2 + 128*zeta3) / 384._default
associate (data => sf_int%data)
    eps = sf_int%data%eps
    x = sf_int%x
    xb = 1 - x
    !!! JRR: WK please check: Depending on the map flag
    !!! r, rb are equal to x, xb or mapped. The function
    !!! value is either 1 for the mapped case (the 0th order
    !!! SF value has been taken into account in the Jacobian)
    !!! or the zeroth order SF value for non-mapped case
    if (sf_int%data%map) then
        rb = xb**eps
        r = 1 - rb
        f = 1
    else
        r = x
        rb = xb
        f = xb ** (-1 + eps)
    end if
    if (data%order > 0) then
        f = f * (1 + g1 * eps)
        x_2 = x*x
        if (rb>0) f = f * (1 - (1-x_2) / (2 * rb))
        if (data%order > 1) then
            f = f * (1 + g2 * eps**2)
            if (rb>0 .and. xb>0 .and. x>xmin) then
                log_x = log (x)
                log_xb = log (xb)
                f = f * (1 - ((1+3*x_2)*log_x + xb * (4*(1+x)*log_xb + 5 + x)) &
                    / ( 8 * rb) * eps)
            end if
            if (data%order > 2) then
                f = f * (1 + g3 * eps**3)
                if (rb > 0 .and. xb > 0 .and. x > xmin) then
                    f = f * (1 - ((1+x) * xb &
                        * (6 * Li2(x) + 12 * log_xb**2 - 3 * pi**2) &
                        + 1.5_default * (1 + 8*x + 3*x_2) * log_x &
                        + 6 * (x+5) * xb * log_xb &
                        + 12 * (1+x_2) * log_x * log_xb &
                        - (1 + 7*x_2) * log_x**2 / 2 &
                        + (39 - 24*x - 15*x_2) / 4) &
                        / ( 48 * rb) * eps**2)
                end if
            end if
        end if
    end if
end if

```

```

        end if
    end associate
    call interaction_set_matrix_element &
        (sf_int%interaction_t, cmplx (f, kind=default))
    sf_int%status = SF_EVALUATED
end subroutine isr_apply

```

### 10.4.7 Unit tests

```

<SF isr: public>+≡
    public :: sf_isr_test

<SF isr: tests>≡
    subroutine sf_isr_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
        !!! JRR: WK please check
        !!! there are probably more tests necessary. e.g. to see whether
        !!! integrals with and without mapping are the same, right?
    <SF isr: execute tests>
    end subroutine sf_isr_test

```

### Test structure function data

Construct and display a test structure function data object.

```

<SF isr: execute tests>≡
    call test (sf_isr_1, "sf_isr_1", &
        "structure function configuration", &
        u, results)

<SF isr: tests>+≡
    subroutine sf_isr_1 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable :: data

        write (u, "(A)")  "* Test output: sf_isr_1"
        write (u, "(A)")  "*   Purpose: initialize and display &
            &test structure function data"
        write (u, "(A)")

        write (u, "(A)")  "* Create empty data object"
        write (u, "(A)")

        call os_data_init (os_data)
        call syntax_model_file_init ()
        call model_list_read_model (var_str ("QED"), &
            var_str ("QED.mdl"), os_data, model)
        call flavor_init (flv, ELECTRON, model)
    end subroutine sf_isr_1

```



```

allocate (isr_data_t :: data)
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize"
write (u, "(A)")

select type (data)
type is (isr_data_t)
    call data%init (model, flv, 1./137._default, 10._default, &
        0.000511_default, order = 3, recoil = .false.)
end select

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)") "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))") data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_isr_1"

end subroutine sf_isr_1

```

## Test and probe structure function

Construct and display a structure function object based on the ISR structure function.

```

<SF isr: execute tests>+≡
    call test (sf_isr_2, "sf_isr_2", &
        "structure function instance", &
        u, results)

<SF isr: tests>+≡
subroutine sf_isr_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, x
    real(default) :: f, s

    write (u, "(A)")  "* Test output: sf_isr_2"
    write (u, "(A)")  "* Purpose: initialize and fill &
        &test structure function object"

```

```

write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("QED"), &
    var_str ("QED.mdl"), os_data, model)
call flavor_init (flv, ELECTRON, model)

call reset_interaction_counter ()

allocate (isr_data_t :: data)
select type (data)
type is (isr_data_t)
    call data%init (model, flv, 1./137._default, 500._default, &
        0.000511_default, order = 3, recoil = .false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, no ISR mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0.5_default
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")

```

```

write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F12.9))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

call sf_int%final ()
deallocate (sf_int)
deallocate (x)
deallocate (r)
deallocate (data)

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (isr_data_t :: data)
select type (data)
type is (isr_data_t)
    call data%init (model, flv, 1./137._default, 500._default, &
        0.000511_default, order = 3, recoil = .true.)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)

```

```

call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.25, with ISR mapping, "
write (u, "(A)")  "          non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.true.)

write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

```

```
call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_isr_2"

end subroutine sf_isr_2
```

## 10.5 EPA

```

⟨sf_epa.f90⟩≡
  ⟨File header⟩

  module sf_epa

    ⟨Use kinds⟩
    ⟨Use strings⟩
    use constants, only: pi !NODEP!
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base

    ⟨Standard module head⟩

    ⟨SF epa: public⟩

    ⟨SF epa: parameters⟩

    ⟨SF epa: types⟩

    contains

    ⟨SF epa: procedures⟩

    ⟨SF epa: tests⟩

  end module sf_epa

```

### 10.5.1 Physics

The EPA structure function for a photon inside an (elementary) particle  $p$  with energy  $E$ , mass  $m$  and charge  $q_p$  (e.g., electron) is given by ( $\bar{x} \equiv 1 - x$ )

$$\begin{aligned}
 f(x) = \frac{\alpha}{\pi} q_p^2 \frac{1}{x} & \left[ \left( \bar{x} + \frac{x^2}{2} \right) \ln \frac{Q_{\max}^2}{Q_{\min}^2} \right. \\
 & \left. - \left( 1 - \frac{x}{2} \right)^2 \ln \frac{x^2 + \frac{Q_{\max}^2}{E^2}}{x^2 + \frac{Q_{\min}^2}{E^2}} - x^2 \frac{m^2}{Q_{\min}^2} \left( 1 - \frac{Q_{\min}^2}{Q_{\max}^2} \right) \right]. \quad (10.46)
 \end{aligned}$$

If no explicit  $Q$  bounds are provided, the kinematical bounds are

$$-Q_{\max}^2 = t_0 = -2\bar{x}(E^2 + p\bar{p}) + 2m^2 \approx -4\bar{x}E^2, \quad (10.47)$$

$$-Q_{\min}^2 = t_1 = -2\bar{x}(E^2 - p\bar{p}) + 2m^2 \approx -\frac{x^2}{\bar{x}}m^2. \quad (10.48)$$

The second and third terms in (10.46) are negative definite (and subleading). Noting that  $\bar{x} + x^2/2$  is bounded between  $1/2$  and  $1$ , we derive that  $f(x)$  is always smaller than

$$\bar{f}(x) = \frac{\alpha}{\pi} q_p^2 \frac{L - 2 \ln x}{x} \quad \text{where} \quad L = \ln \frac{\min(4E_{\max}^2, Q_{\max}^2)}{\max(m^2, Q_{\min}^2)}, \quad (10.49)$$

where we allow for explicit  $Q$  bounds that narrow the kinematical range. Therefore, we generate this distribution:

$$\int_{x_0}^{x_1} dx \bar{f}(x) = C(x_0, x_1) \int_0^1 dx' \quad (10.50)$$

We set

$$\ln x = \frac{1}{2} \left\{ L - \sqrt{L^2 - 4 [x' \ln x_1 (L - \ln x_1) + \bar{x}' \ln x_0 (L - \ln x_0)]} \right\} \quad (10.51)$$

such that  $x(0) = x_0$  and  $x(1) = x_1$  and

$$\frac{dx}{dx'} = \left( \frac{\alpha}{\pi} q_p^2 \right)^{-1} x \frac{C(x_0, x_1)}{L - 2 \ln x} \quad (10.52)$$

with

$$C(x_0, x_1) = \frac{\alpha}{\pi} q_p^2 [\ln x_1 (L - \ln x_1) - \ln x_0 (L - \ln x_0)] \quad (10.53)$$

such that (10.50) is satisfied. Finally, we have

$$\int_{x_0}^{x_1} dx f(x) = C(x_0, x_1) \int_0^1 dx' \frac{f(x(x'))}{\bar{f}(x(x'))} \quad (10.54)$$

where  $x'$  is calculated from  $x$  via (10.51).

The structure of the mapping is most obvious from:

$$x'(x) = \frac{\log x (L - \log x) - \log x_0 (L - \log x_0)}{\log x_1 (L - \log x_1) - \log x_0 (L - \log x_0)}. \quad (10.55)$$

### 10.5.2 The EPA data block

The EPA parameters are:  $\alpha$ ,  $E_{\max}$ ,  $m$ ,  $Q_{\min}$ , and  $x_{\min}$ . Instead of  $m$  we can use the incoming particle PDG code as input; from this we can deduce the mass and charge.

Internally we store in addition  $C_{0/1} = \frac{\alpha}{\pi} q_e^2 \ln x_{0/1} (L - \ln x_{0/1})$ , the c.m. energy squared and the incoming particle mass.

```
<SF epa: public>≡
public :: epa_data_t
```

```

<SF epa: types>≡
  type, extends(sf_data_t) :: epa_data_t
  private
    type(model_t), pointer :: model => null ()
    type(flavor_t) :: flv
    real(default) :: alpha
    real(default) :: x_min
    real(default) :: x_max
    real(default) :: q_min
    real(default) :: q_max
    real(default) :: E_max
    real(default) :: mass
    real(default) :: log
    real(default) :: c0
    real(default) :: c1
    integer :: error = NONE
    logical :: recoil = .false.
    logical :: map = .true.
  contains
    <SF epa: epa data: TBP>
  end type epa_data_t

```

Error codes

```

<SF epa: parameters>≡
  integer, parameter :: NONE = 0
  integer, parameter :: ZERO_QMIN = 1
  integer, parameter :: Q_MAX_TOO_SMALL = 2
  integer, parameter :: ZERO_XMIN = 3

<SF epa: epa data: TBP>≡
  procedure :: init => epa_data_init

<SF epa: procedures>≡
  subroutine epa_data_init &
    (data, model, flv, alpha, x_min, q_min, E_max, mass, recoil)
    class(epa_data_t), intent(inout) :: data
    type(model_t), intent(in), target :: model
    type(flavor_t), intent(in) :: flv
    real(default), intent(in) :: alpha, x_min, q_min, E_max
    real(default), intent(in), optional :: mass
    logical, intent(in), optional :: recoil
    data%model => model
    data%flv = flv
    data%alpha = alpha
    data%E_max = E_max
    data%x_min = x_min
    data%x_max = 1
    if (data%x_min == 0) then
      data%error = ZERO_XMIN; return
    end if
    data%q_min = q_min
    data%q_max = 2 * data%E_max
    select case (char(model_get_name(data%model)))
    case ("QCD", "Test")
      call msg_fatal ("EPA structure function not available for model " &

```



```

        // char (model_get_name (data%model)) // ".")
end select
if (present (recoil)) then
    data%recoil = recoil
end if
if (present (mass)) then
    data%mass = mass
else
    data%mass = flavor_get_mass (flv)
end if
if (max (data%mass, data%q_min) == 0) then
    data%error = ZERO_QMIN; return
else if (max (data%mass, data%q_min) >= data%E_max) then
    data%error = Q_MAX_TOO_SMALL; return
end if
data%log = log (4 * (data%E_max / max (data%mass, data%q_min)) ** 2 )
data%c0 = log (data%x_min) * (data%log - log (data%x_min))
data%c1 = log (data%x_max) * (data%log - log (data%x_max))
end subroutine epa_data_init

```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

```

<SF epa: epa data: TBP>+≡
    procedure :: check => epa_data_check

<SF epa: procedures>+≡
    subroutine epa_data_check (data)
        class(epa_data_t), intent(in) :: data
        select case (data%error)
            case (ZERO_QMIN)
                call msg_fatal (" EPA: Particle mass is zero")
            case (Q_MAX_TOO_SMALL)
                call msg_fatal (" EPA: Particle mass exceeds Qmax")
            case (ZERO_XMIN)
                call msg_fatal (" EPA: x_min must be larger than zero")
        end select
    end subroutine epa_data_check

```

Output

```

<SF epa: epa data: TBP>+≡
    procedure :: write => epa_data_write

<SF epa: procedures>+≡
    subroutine epa_data_write (data, unit, verbose) !, md5)
        class(epa_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        !!! JRR: please check: do we still need this?
    !    logical, intent(in), optional :: md5
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "EPA data:"
        if (flavor_is_defined (data%flv)) then
            write (u, "(3x,A)", advance="no") " flavor = "

```

```

        call flavor_write (data%flv, u); write (u, *)
        write (u, "(3x,A,ES19.12)") " alpha = ", data%alpha
        write (u, "(3x,A,ES19.12)") " x_min = ", data%x_min
        write (u, "(3x,A,ES19.12)") " x_max = ", data%x_max
        write (u, "(3x,A,ES19.12)") " q_min = ", data%q_min
        write (u, "(3x,A,ES19.12)") " q_max = ", data%q_max
        write (u, "(3x,A,ES19.12)") " E_max = ", data%e_max
        write (u, "(3x,A,ES19.12)") " mass = ", data%mass
        write (u, "(3x,A,ES19.12)") " log = ", data%log
        write (u, "(3x,A,ES19.12)") " c0 = ", data%c0
        write (u, "(3x,A,ES19.12)") " c1 = ", data%c1
        write (u, "(3x,A,L2)") " recoil = ", data%recoil
        write (u, "(3x,A,L2)") " map = ", data%map
    else
        write (u, "(3x,A)") "[undefined]"
    end if
end subroutine epa_data_write

```

The number of kinematic parameters.

```

<SF epa: epa data: TBP>+≡
    procedure :: get_n_par => epa_data_get_n_par

<SF epa: procedures>+≡
    function epa_data_get_n_par (data) result (n)
        class(epa_data_t), intent(in) :: data
        integer :: n
        if (data%recoil) then
            n = 3
        else
            n = 1
        end if
    end function epa_data_get_n_par

```

Return the outgoing particles PDG codes. The outgoing particle is always the photon while the radiated particle is identical to the incoming one.

```

<SF epa: epa data: TBP>+≡
    procedure :: get_pdg_out => epa_data_get_pdg_out

<SF epa: procedures>+≡
    !!! JRR: WK please check
    function epa_data_get_pdg_out (data) result (pdg_out)
        class(epa_data_t), intent(in) :: data
        integer, dimension(:), allocatable :: pdg_out
        integer :: i, n
        n = 1
        allocate (pdg_out (n))
        pdg_out(1) = PHOTON
    end function epa_data_get_pdg_out

```

Allocate the interaction record.

```

<SF epa: epa data: TBP>+≡
    procedure :: allocate_sf_int => epa_data_allocate_sf_int

```

```

<SF epa: procedures>+≡
  subroutine epa_data_allocate_sf_int (data, sf_int)
    class(epa_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (epa_t :: sf_int)
  end subroutine epa_data_allocate_sf_int

```

### 10.5.3 The EPA object

The `epa_t` data type is a  $1 \rightarrow 2$  interaction. We should be able to handle several flavors in parallel, since EPA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and  $q_{\min}$  applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, photon), where the photon initiates the hard interaction.

We generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way. !!! JRR: WK please check For EPA we don't seem to need the `q` variable.

```

<SF epa: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: epa_t
    type(epa_data_t), pointer :: data => null ()
    real(default) :: x = 0
    real(default) :: q = 0
  contains
    <SF epa: epa: TBP>
  end type epa_t

```

Type string: has to here, but there is no string variable on which EPA depends. Hence, a dummy routine.

```

<SF epa: epa: TBP>≡
  procedure :: type_string => epa_type_string

<SF epa: procedures>+≡
  function epa_type_string (object) result (string)
    class(epa_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "EPA: equivalent photon approx."
    else
      string = "EPA: [undefined]"
    end if
  end function epa_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF epa: epa: TBP>+≡
  procedure :: write => epa_write

```

```

<SF epa: procedures>+≡
subroutine epa_write (object, unit)
  !!! JRR: WK please check
  class(epa_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
    call object%data%write (u)
    if (object%status >= SF_DONE_KINEMATICS) then
      write (u, "(1x,A)") "SF parameters:"
      write (u, "(3x,A,ES17.10)") "x =", object%x
      if (object%status >= SF_FAILED_EVALUATION) then
        write (u, "(3x,A,ES17.10)") "Q =", object%q
      end if
    end if
    call object%base_write (u)
  else
    write (u, "(1x,A)") "EPA data: [undefined]"
  end if
end subroutine epa_write

```

#### 10.5.4 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set (which is the default), we are asked to provide an efficient mapping. This has been written down above and uses the upper limit on the structure function.

```

<SF epa: epa: TBP>+≡
procedure :: complete_kinematics => epa_complete_kinematics

<SF epa: procedures>+≡
subroutine epa_complete_kinematics (sf_int, x, f, r, map)
  !!! JRR: WK please check
  class(epa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  logical, intent(in) :: map
  real(default) :: rb1, xb1
  real(default) :: d, lx
  if (map) then
    rb1 = 1 - r(1)
    d = sf_int%data%log ** 2 - 4 * (r(1) * sf_int%data%c1 &
      + rb1 * sf_int%data%c0)
    if (d <= 0) then
      sf_int%status = SF_FAILED_KINEMATICS
      f = 0
      return
    else
      lx = (sf_int%data%log - sqrt (d)) / 2
    end if
  end if
end subroutine epa_complete_kinematics

```

```

        end if
        x(1) = exp (lx)
        f = x(1) * (sf_int%data%c1 - sf_int%data%c0) / (sf_int%data%log - 2 * lx)
    else
        x(1) = r(1)
        if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
            f = 1
        else
            sf_int%status = SF_FAILED_KINEMATICS
            f = 0
            return
        end if
    end if
    end if
    xb1 = 1 - x(1)
    if (size(x) == 3) x(2:3) = r(2:3)
    !!! JRR: WK please check
    !!! in kinematics we have to set the map flag in order
    !!! to be available in apply
    sf_int%data%map = map
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end subroutine epa_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF epa: epa: TBP>+≡
    procedure :: inverse_kinematics => epa_inverse_kinematics

<SF epa: procedures>+≡
    subroutine epa_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        !!! JRR: WK please check
        class(epa_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
        real(default) :: xb1
        real(default) :: lx
        logical :: set_mom
        set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
        if (map) then
            lx = log (x(1))
            r(1) = (lx * (sf_int%data%log - lx) - sf_int%data%c0) / &
                (sf_int%data%c1 - sf_int%data%c0)
            f = x(1) * (sf_int%data%c1 - sf_int%data%c0) / (sf_int%data%log - 2 * lx)
        else

```

```

r(1) = x(1)
if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
  f = 1
else
  f = 0
end if
end if
xb1 = 1 - x(1)
if (size(r) == 3) r(2:3) = x(2:3)
if (set_mom) then
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
    sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
    sf_int%x = 0
    f = 0
  end select
end if
end subroutine epa_inverse_kinematics

```

$\langle SF \text{ epa: epa: TBP} \rangle + \equiv$

```

procedure :: init => epa_init

```

$\langle SF \text{ epa: procedures} \rangle + \equiv$

```

subroutine epa_init (sf_int, data)
  class(epa_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t) :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
  type(flavor_t) :: flv_photon
  type(quantum_numbers_t) :: qn_photon, qn
  type(state_iterator_t) :: it_hel
  mask = new_quantum_numbers_mask (.false., .false., &
    mask_h = (/ .false., .false., .true. /))
  hel_lock = (/ 2, 1, 0 /)
  select type (data)
  type is (epa_data_t)
    call sf_int%base_init (mask, [data%mass**2], &
      [data%mass**2], [0._default], hel_lock = hel_lock)
    sf_int%data => data
    call flavor_init (flv_photon, PHOTON, data%model)
    call quantum_numbers_init (qn_photon, flv_photon)
    call polarization_init_generic (pol, data%flv)
    call quantum_numbers_init (qn_fc(1), &
      flv = data%flv, col = color_from_flavor (data%flv, 1))
    call state_iterator_init (it_hel, pol%state)
    do while (state_iterator_is_valid (it_hel))
      qn_hel = state_iterator_get_quantum_numbers (it_hel)
      qn = qn_hel(1) .merge. qn_fc(1)
      call interaction_add_state (sf_int%interaction_t, &
        (/ qn, qn, qn_photon /))
    end do
  end select
end subroutine

```

```

        call state_iterator_advance (it_hel)
    end do
    call polarization_final (pol)
    call interaction_freeze (sf_int%interaction_t)
    !!! JRR: WK please check
    call sf_int%set_incoming ([1])
    call sf_int%set_radiated ([2])
    call sf_int%set_outgoing ([3])
    sf_int%status = SF_INITIAL
end select
end subroutine epa_init

```

### 10.5.5 EPA structure function

The EPA structure function allows for a straightforward mapping of the unit interval. The  $x$  value is transformed, and the mapped structure function becomes unity at its upper boundary.

The structure function implementation applies the above mapping to the input (random) number  $r$  to generate the momentum fraction  $x$  and the function value  $f$ . For numerical stability reasons, we also output  $xb$ , which is  $\bar{x} = 1 - x$ .

If `map` is set, the mapping is applied, otherwise it is switched off.

(EPA: procedures)≡

```

elemental subroutine strfun (f, x, xb, r, E, data, no_map)
    real(default), intent(out) :: f, x, xb
    real(default), intent(in)  :: r, E
    type(epa_data_t), intent(in) :: data
    logical, intent(in) :: no_map
    real(default) :: rb, lx0, lx1, lx, d, den
    real(default) :: qmaxsq, qminsq
    f = 0
    rb = 1 - r
    if (no_map) then
        x = r
        if (data%x_min < x .and. x < data%x_max) then
            den = (log (data%x_max) * (data%log - log (data%x_max)) &
                  - log (data%x_min) * (data%log - log (data%x_min))) / x
        else
            den = 0
        end if
    else
        lx0 = log (data%x_min)
        lx1 = log (data%x_max)
        d = data%log ** 2 &
          - 4 * (r * lx1 * (data%log - lx1) + rb * lx0 * (data%log - lx0))
        if (d <= 0) then
            return
        else
            lx = (data%log - sqrt (d)) / 2
        end if
        x = exp (lx)
        den = data%log - 2 * lx
    end if

```

```

if (den <= 0) return
xb = 1 - x
qminsq = max (x ** 2 / xb * data%mass ** 2, data%q_min ** 2)
qmaxsq = min (4 * E ** 2, data%q_max ** 2)
if (qminsq < qmaxsq) then
  f = ((xb + x ** 2 / 2) * log (qmaxsq / qminsq) &
    - (1 - x / 2) ** 2 &
    * log ((x**2 + qmaxsq / E ** 2) / (x**2 + qminsq / E ** 2))) &
    - x ** 2 * data%mass ** 2 / qminsq * (1 - qminsq / qmaxsq)) &
    * ( data%c1 - data%c0 ) / den
end if
end subroutine strfun

```

### 10.5.6 EPA application

For EPA, we can in principle compute kinematics and function value in a single step. In order to be able to reweight events, kinematics and structure function application are separated. This function works on a single beam, assuming that the input momentum has been set. We need three random numbers as input: one for  $x$ , and two for the polar and azimuthal angles. Alternatively, for the no-recoil case, we can skip  $p_T$  generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated  $x$  values are identical.

```

<SF epa: epa: TBP>+≡
  procedure :: apply => epa_apply

<SF epa: procedures>+≡
  subroutine epa_apply (sf_int, scale) !!!r, epa_data, no_map)
    !!! JRR: WK please check
    class(epa_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default) :: r, rb
    real(default) :: x, xb, qminsq, qmaxsq, f
    associate (data => sf_int%data)
      x = sf_int%x
      f = 0
      xb = 1 - x
      !!! JRR: WK please check: should be correct. The structure
      !!! function depends on mapped variables in the mapped case
      !!! and the true r values (constrained to interval [xmin, xmax])
      !!! for the unmapped case. Jacobian is taken into account in
      !!! complete_kinematics. Is this true also for the qminsq bound
      !!! or do we have to use the unmapped x values there?
      qminsq = max (x ** 2 / xb * data%mass ** 2, data%q_min ** 2)
      qmaxsq = min (4 * scale ** 2, data%q_max ** 2)
      if (qminsq < qmaxsq) then
        f = (data%alpha / PI) * flavor_get_charge (data%flv)**2 * &
          ((xb + x ** 2 / 2) * log (qmaxsq / qminsq) &
            - (1 - x / 2) ** 2 * log &
            ((x**2 + qmaxsq / scale ** 2) / (x**2 + qminsq / scale ** 2))) &
            - x ** 2 * data%mass ** 2 / qminsq * (1 - qminsq / qmaxsq))
      end if
    end associate
  end subroutine epa_apply

```



```

        end if
    end associate
    call interaction_set_matrix_element &
        (sf_int%interaction_t, cmplx (f, kind=default))
    sf_int%status = SF_EVALUATED
end subroutine epa_apply

```

### 10.5.7 Unit tests

```

<SF epa: public>+≡
    public :: sf_epa_test

<SF epa: tests>≡
    subroutine sf_epa_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
        !!! JRR: WK please check
        !!! there are probably more tests necessary. e.g. to see whether
        !!! integrals with and without mapping are the same, right?
    <SF epa: execute tests>
    end subroutine sf_epa_test

```

#### Test structure function data

Construct and display a test structure function data object.

```

<SF epa: execute tests>≡
    call test (sf_epa_1, "sf_epa_1", &
        "structure function configuration", &
        u, results)

<SF epa: tests>+≡
    subroutine sf_epa_1 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable :: data

        write (u, "(A)")  "* Test output: sf_epa_1"
        write (u, "(A)")  "* Purpose: initialize and display &
            &test structure function data"
        write (u, "(A)")

        write (u, "(A)")  "* Create empty data object"
        write (u, "(A)")

        call os_data_init (os_data)
        call syntax_model_file_init ()
        call model_list_read_model (var_str ("QED"), &
            var_str ("QED.mdl"), os_data, model)
        call flavor_init (flv, ELECTRON, model)
    end subroutine sf_epa_1

```

```

allocate (epa_data_t :: data)
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize"
write (u, "(A)")

select type (data)
type is (epa_data_t)
    call data%init (model, flv, 1./137._default, 0.01_default, &
        10._default, 50._default, 0.000511_default, recoil = .false.)
end select

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)") "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))") data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_epa_1"

end subroutine sf_epa_1

```

## Test and probe structure function

Construct and display a structure function object based on the EPA structure function.

```

<SF epa: execute tests>+≡
    call test (sf_epa_2, "sf_epa_2", &
        "structure function instance", &
        u, results)

<SF epa: tests>+≡
subroutine sf_epa_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, x
    real(default) :: f, s

    write (u, "(A)")  "* Test output: sf_epa_2"
    write (u, "(A)")  "* Purpose: initialize and fill &
        &test structure function object"

```

```

write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("QED"), &
    var_str ("QED.mdl"), os_data, model)
call flavor_init (flv, ELECTRON, model)

call reset_interaction_counter ()

allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
    call data%init (model, flv, 1./137._default, 0.01_default, &
        10._default, 50._default, 0.000511_default, recoil = .false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, no EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0.5_default
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")

```

```

write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F12.9))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate EPA structure function"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

call sf_int%final ()
deallocate (sf_int)
deallocate (x)
deallocate (r)
deallocate (data)

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
    call data%init (model, flv, 1./137._default, 0.01_default, &
        10._default, 50._default, 0.000511_default, recoil = .true.)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)

```

```

call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.25, with EPA mapping, "
write (u, "(A)")  "          non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, map=.true.)

write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Evaluate EPA structure function"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

```

```
call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_epa_2"

end subroutine sf_epa_2
```

## 10.6 EWA

```
<sf_ewa.f90>≡  
  <File header>  
  
  module sf_ewa  
  
    <Use kinds>  
    <Use strings>  
    <Use file utils>  
    use constants, only: pi !NODEP!  
    use diagnostics !NODEP!  
    use lorentz !NODEP!  
    use unit_tests  
    use os_interface  
    use models  
    use flavors  
    use colors  
    use quantum_numbers  
    use state_matrices  
    use polarizations  
    use interactions  
    use sf_aux  
    use sf_base  
  
    <Standard module head>  
  
    <SF ewa: public>  
  
    <SF ewa: parameters>  
  
    <SF ewa: types>  
  
    contains  
  
    <SF ewa: procedures>  
  
    <SF ewa: tests>  
  
  end module sf_ewa
```

### 10.6.1 Physics

The EWA structure function for a  $Z$  or  $W$  inside a fermion (lepton or quark) depends on the vector-boson polarization. We distinguish transversal ( $\pm$ ) and

longitudinal (0) polarization.

$$F_+(x) = \frac{1}{16\pi^2} \frac{(v-a)^2 + (v+a)^2 \bar{x}^2}{x} \left[ \ln \left( \frac{p_{\perp,\max}^2 + \bar{x}M^2}{\bar{x}M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2} \right] \quad (10.56)$$

$$F_-(x) = \frac{1}{16\pi^2} \frac{(v+a)^2 + (v-a)^2 \bar{x}^2}{x} \left[ \ln \left( \frac{p_{\perp,\max}^2 + \bar{x}M^2}{\bar{x}M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2} \right] \quad (10.57)$$

$$F_0(x) = \frac{v^2 + a^2}{8\pi^2} \frac{2\bar{x}}{x} \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2} \quad (10.58)$$

where  $p_{\perp,\max}$  is the cutoff in transversal momentum,  $M$  is the vector-boson mass,  $v$  and  $a$  are the vector and axial-vector couplings, and  $\bar{x} \equiv 1-x$ . Note that the longitudinal structure function is finite for large cutoff, while the transversal structure function is logarithmically divergent.

The maximal transverse momentum is given by the kinematical limit, it is

$$p_{\perp,\max} = \bar{x}\sqrt{s}/2. \quad (10.59)$$

The vector and axial couplings for a fermion branching into a  $W$  are

$$v_W = \frac{g}{2\sqrt{2}}, \quad a_W = \frac{g}{2\sqrt{2}}. \quad (10.60)$$

For  $Z$  emission, this is replaced by

$$v_Z = \frac{g}{2\cos\theta_w} (t_3 - 2q\sin^2\theta_w), \quad a_Z = \frac{g}{2\cos\theta_w} t_3, \quad (10.61)$$

where  $t_3 = \pm\frac{1}{2}$  is the fermion isospin, and  $q$  its charge.

For an initial antifermion, the signs of the axial couplings are inverted. Note that a common sign change of  $v$  and  $a$  is irrelevant.

The EWA depends on the parameters  $g$ ,  $\sin^2\theta_w$ ,  $M_W$ , and  $M_Z$ . These can all be taken from the SM input, and the prefactors are calculated from those and the incoming particle type.

Since these structure functions have a  $1/x$  singularity (which is not really relevant in practice, however, since the vector boson mass is finite), we map this singularity allowing for nontrivial  $x$  bounds:

$$x = \exp(\bar{r} \ln x_0 + r \ln x_1) \quad (10.62)$$

such that

$$\int_{x_0}^{x_1} \frac{dx}{x} = (\ln x_1 - \ln x_0) \int_0^1 dr. \quad (10.63)$$

As a user parameter, we have the cutoff  $p_{\perp,\max}$ . The divergence  $1/x$  also requires a  $x_0$  cutoff; and for completeness we introduce a corresponding  $x_1$ . Physically, the minimal sensible value of  $x$  is  $M^2/s$ , although the approximation loses its value already at higher  $x$  values.



## 10.6.2 The EWA data block

The EWA parameters are:  $p_{T,\max}$ ,  $c_V$ ,  $c_A$ , and  $m$ . Instead of  $m$  we can use the incoming particle PDG code as input; from this we can deduce the mass and charges. In the initialization phase it is not yet determined whether a  $W$  or a  $Z$  is radiated, hence we set the vector and axial-vector couplings equal to the common prefactors  $g/2 = e/2/\sin\theta_W$ .

In principle, for EWA it would make sense to allow the user to also set the upper bound for  $x$ ,  $x_{\max}$ , but we fix it to one here.

```

<SF ewa: public>≡
  public :: ewa_data_t

<SF ewa: types>≡
  type, extends(sf_data_t) :: ewa_data_t
  private
    type(model_t), pointer :: model => null ()
    type(flavor_t) :: flv
    real(default) :: pt_max
    real(default) :: sqrts
    real(default) :: x_min
    real(default) :: x_max
    real(default) :: mass
    real(default) :: q_min
    real(default) :: cv
    real(default) :: ca
    real(default) :: costhw
    real(default) :: sinthw
    real(default) :: mW
    real(default) :: mZ
    real(default) :: coeff
    logical :: keep_momentum
    logical :: keep_energy
    integer :: id = 0
    integer :: error = 0
    logical :: map = .true.
  contains
    <SF ewa: ewa data: TBP>
  end type ewa_data_t

```

Error codes

```

<SF ewa: parameters>≡
  integer, parameter :: NONE = 0
  integer, parameter :: ZERO_QMIN = 1
  integer, parameter :: Q_MAX_TOO_SMALL = 2
  integer, parameter :: ZERO_XMIN = 3

<SF ewa: ewa data: TBP>≡
  procedure :: init => ewa_data_init

<SF ewa: procedures>≡
  subroutine ewa_data_init (data, model, flv, x_min, q_min, pt_max, &
    sqrts, keep_momentum, keep_energy, mass)
    class(ewa_data_t), intent(inout) :: data
    type(model_t), intent(in), target :: model
    type(flavor_t), intent(in) :: flv

```

```

real(default), intent(in) :: x_min, q_min, pt_max, sqrts
logical, intent(in) :: keep_momentum, keep_energy
real(default), intent(in), optional :: mass
real(default) :: g, ee, sinthw
data%model => model
data%flv = flv
data%pt_max = pt_max
data%sqrts = sqrts
data%x_min = x_min
data%x_max = 1
if (data%x_min == 0) then
    data%error = ZERO_XMIN; return
end if
select case (char (model_get_name (data%model)))
case ("QCD", "QED", "Test")
    call msg_fatal ("EWA structure function not available for model " &
        // char (model_get_name (data%model)) // ".")
end select
ee = model_get_parameter_value (data%model, var_str ("ee"))
data%sinthw = model_get_parameter_value (data%model, var_str ("sw"))
data%costhw = model_get_parameter_value (data%model, var_str ("cw"))
data%mZ = model_get_parameter_value (data%model, var_str ("mZ"))
data%mW = model_get_parameter_value (data%model, var_str ("mW"))
if (data%sinthw /= 0) then
    g = ee / data%sinthw
else
    call msg_fatal ("Vanishing value of sin(theta_w).")
end if
data%cv = g / 2._default
data%ca = g / 2._default
data%coeff = 1._default / (8._default * PI**2)
data%keep_momentum = keep_momentum
data%keep_energy = keep_energy
if (present (mass)) then
    data%mass = mass
else
    data%mass = flavor_get_mass (flv)
end if
end subroutine ewa_data_init

```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

```

<SF ewa: ewa data: TBP>+≡
    procedure :: check => ewa_data_check

<SF ewa: procedures>+≡
    subroutine ewa_data_check (data)
        class(ewa_data_t), intent(in) :: data
        select case (data%error)
        case (ZERO_QMIN)
            call msg_fatal (" EWA: Particle mass is zero")
        case (Q_MAX_TOO_SMALL)
            call msg_fatal (" EWA: Particle mass exceeds Qmax")
        case (ZERO_XMIN)

```

```

        call msg_fatal (" EWA: x_min must be larger than zero")
    end select
end subroutine ewa_data_check

```

Set the vector boson ID for distinguishing  $W$  and  $Z$  bosons.

```

<SF ewa: ewa data: TBP>+≡
    !!! JRR: WK please check. This has to be set somewhere in the
    !!! new structure. I have no clue, where. :(
    procedure :: set_id => ewa_set_id

<SF ewa: procedures>+≡
    subroutine ewa_set_id (data, id)
        class(ewa_data_t), intent(inout) :: data
        integer, intent(in) :: id
        data%id = id
    end subroutine ewa_set_id

```

Output

```

<SF ewa: ewa data: TBP>+≡
    procedure :: write => ewa_data_write

<SF ewa: procedures>+≡
    subroutine ewa_data_write (data, unit, verbose) !, md5)
        class(ewa_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        !!! JRR: WK please check: still needed?
    ! logical, intent(in), optional :: md5
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "EWA data:"
        if (flavor_is_defined (data%flv)) then
            write (u, "(3x,A)", advance="no") " flavor = "
            call flavor_write (data%flv, u); write (u, *)
            write (u, "(3x,A,ES19.12)") " x_min = ", data%x_min
            write (u, "(3x,A,ES19.12)") " x_max = ", data%x_max
            write (u, "(3x,A,ES19.12)") " pt_max = ", data%pt_max
            write (u, "(3x,A,ES19.12)") " sqrts = ", data%sqrts
            write (u, "(3x,A,ES19.12)") " mass = ", data%mass
            write (u, "(3x,A,ES19.12)") " cv = ", data%cv
            write (u, "(3x,A,ES19.12)") " ca = ", data%ca
            write (u, "(3x,A,ES19.12)") " coeff = ", data%coeff
            write (u, "(3x,A,ES19.12)") " costhw = ", data%costhw
            write (u, "(3x,A,ES19.12)") " sinthw = ", data%sinthw
            write (u, "(3x,A,ES19.12)") " mZ = ", data%mZ
            write (u, "(3x,A,ES19.12)") " mW = ", data%mW
            write (u, "(3x,A,L2)") " keep_mom. = ", data%keep_momentum
            write (u, "(3x,A,L2)") " keep_en. = ", data%keep_energy
            write (u, "(3x,A,L2)") " map = ", data%map
        else
            write (u, "(3x,A)") "[undefined]"
        end if
    end subroutine ewa_data_write

```

The number of parameters is one for collinear splitting, in case one of the options `keep_energy` or `keep_momentum` is set, we take the recoil into account.

```

<SF ewa: ewa data: TBP>+≡
  procedure :: get_n_par => ewa_data_get_n_par

<SF ewa: procedures>+≡
  function ewa_data_get_n_par (data) result (n)
    class(ewa_data_t), intent(in) :: data
    integer :: n
    if (data%keep_energy .or. data%keep_momentum) then
      n = 3
    else
      n = 1
    end if
  end function ewa_data_get_n_par

```

Return the outgoing particles PDG codes. This depends, whether this is a charged-current or neutral-current interaction.

```

<SF ewa: ewa data: TBP>+≡
  procedure :: get_pdg_out => ewa_data_get_pdg_out

<SF ewa: procedures>+≡
  !!! JRR: WK please check
  function ewa_data_get_pdg_out (data) result (pdg_out)
    class(ewa_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    real(default) :: t3
    logical :: up_type
    integer :: i, n
    n = 1
    allocate (pdg_out (n))
    t3 = - flavor_get_isospin (data%flv)
    up_type = (t3 > 0)
    select case (data%id)
      case(23)
        pdg_out(1) = flavor_get_pdg (data%flv)
      case(24)
        if (up_type) then
          pdg_out(1) = flavor_get_pdg (data%flv) - 1
        else
          pdg_out(1) = flavor_get_pdg (data%flv) + 1
        end if
      end select
    end function ewa_data_get_pdg_out

```

Allocate the interaction record.

```

<SF ewa: ewa data: TBP>+≡
  procedure :: allocate_sf_int => ewa_data_allocate_sf_int

<SF ewa: procedures>+≡
  subroutine ewa_data_allocate_sf_int (data, sf_int)
    class(ewa_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (ewa_t :: sf_int)

```

```
end subroutine ewa_data_allocate_sf_int
```

### 10.6.3 The EWA object

The `ewa_t` data type is a  $1 \rightarrow 2$  interaction. We should be able to handle several flavors in parallel, since EWA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and  $q_{\min}$  applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, W/Z), where the W/Z initiates the hard interaction.

For EPA, we generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way. I do not know whether the same can/should be done for EWA, as the structure functions depend on the W/Z polarization. If we are having Z bosons, both up- and down-type fermions can participate. Otherwise, with a  $W^+$  an up-type fermion is transferred to a down-type fermion, and the other way round.

```
<SF ewa: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: ewa_t
    type(ewa_data_t), pointer :: data => null ()
    real(default) :: x = 0
    real(default) :: q = 0
  contains
    <SF ewa: ewa: TBP>
  end type ewa_t
```

Type string: has to here, but there is no string variable on which EWA depends. Hence, a dummy routine.

```
<SF ewa: ewa: TBP>≡
  procedure :: type_string => ewa_type_string

<SF ewa: procedures>+≡
  function ewa_type_string (object) result (string)
    class(ewa_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "EWA: equivalent W/Z approx."
    else
      string = "EWA: [undefined]"
    end if
  end function ewa_type_string
```

Output. Call the interaction routine after displaying the configuration.

```
<SF ewa: ewa: TBP>+≡
  procedure :: write => ewa_write

<SF ewa: procedures>+≡
  subroutine ewa_write (object, unit)
    !!! JRR: WK please check
    class(ewa_t), intent(in) :: object
```

```

integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit)
if (associated (object%data)) then
  call object%data%write (u)
  if (object%status >= SF_DONE_KINEMATICS) then
    write (u, "(1x,A)") "SF parameters:"
    write (u, "(3x,A,ES17.10)") "x =", object%x
    if (object%status >= SF_FAILED_EVALUATION) then
      write (u, "(3x,A,ES17.10)") "Q =", object%q
    end if
  end if
  call object%base_write (u)
else
  write (u, "(1x,A)") "EWA data: [undefined]"
end if
end subroutine ewa_write

```

*<SF ewa: ewa: TBP>+≡*

```

procedure :: init => ewa_init

```

*<SF ewa: procedures>+≡*

```

subroutine ewa_init (sf_int, data)
  !!! JRR: WK please check
  class(ewa_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t) :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel, qn_fc_fin
  type(flavor_t) :: flv_z, flv_wp, flv_wm, flv_down, flv_up
  type(quantum_numbers_t) :: qn_z, qn_wp, qn_wm, qn, qn_down, &
    qn_up
  type(state_iterator_t) :: it_hel
  real(default) :: q, t3
  real(default) :: m_out
  integer :: i, isospin, pdg
  logical :: up_type
  select type (data)
  type is (ewa_data_t)
    isospin = flavor_get_isospin_type (data%flv)
    pdg = flavor_get_pdg (data%flv)
    if (abs(isospin) /= 2 .or. abs(pdg) > 16 .or. pdg == 0 .or. pdg == 9) then
      call msg_fatal ("EWA structure function only accessible for " &
        // "SM quarks and leptons.")
    end if
    q = - flavor_get_charge (data%flv)
    t3 = - flavor_get_isospin (data%flv)
    up_type = (t3 > 0)
    mask = new_quantum_numbers_mask (.false., .false., &
      mask_h = (/ .false., .false., .true. /))
    hel_lock = (/ 2, 1, 0 /)
    select case (data%id)
    case (23)

```

```

!!! Z boson, flavor is not changing
call sf_int%base_init (mask, [data%mass**2], [data%mass**2], &
    [data%mZ**2], hel_lock = hel_lock)
sf_int%data => data
call flavor_init (flv_z, Z_BOSON, data%model)
call quantum_numbers_init (qn_z, flv_z)
call polarization_init_generic (pol, data%flv)
call quantum_numbers_init (qn_fc(1), &
    flv = data%flv, col = color_from_flavor (data%flv, 1))
call state_iterator_init (it_hel, pol%state)
do while (state_iterator_is_valid (it_hel))
    qn_hel = state_iterator_get_quantum_numbers (it_hel)
    qn = qn_hel(1) .merge. qn_fc(1)
    call interaction_add_state &
        (sf_int%interaction_t, (/ qn, qn, qn_z /))
    call state_iterator_advance (it_hel)
end do
call polarization_final (pol)
case (24)
if (up_type) then
    !!! W+, flavor changing
    call interaction_init (sf_int%interaction_t, &
        1, 0, 2, mask=mask, hel_lock=hel_lock, set_relations=.true.)
    call flavor_init (flv_down, pdg - 1, data%model)
    call flavor_init (flv_wp, W_BOSON, data%model)
    m_out = flavor_get_mass (flv_down)
    call sf_int%base_init (mask, [data%mass**2], [m_out**2], &
        [data%mW**2], hel_lock = hel_lock)
    sf_int%data => data
    call quantum_numbers_init (qn_wp, flv_wp)
    call polarization_init_generic (pol, data%flv)
    call quantum_numbers_init (qn_fc(1), &
        flv = data%flv, col = color_from_flavor (data%flv, 1))
    call quantum_numbers_init (qn_fc_fin(1), &
        flv = flv_down, col = color_from_flavor (flv_down))
    call state_iterator_init (it_hel, pol%state)
    do while (state_iterator_is_valid (it_hel))
        qn_hel = state_iterator_get_quantum_numbers (it_hel)
        qn = qn_hel(1) .merge. qn_fc(1)
        qn_down = qn_hel(1) .merge. qn_fc_fin(1)
        call interaction_add_state &
            (sf_int%interaction_t, (/ qn, qn_down, qn_wp /))
        call state_iterator_advance (it_hel)
    end do
    call polarization_final (pol)
else
    !!! W-, flavor changing
    call flavor_init (flv_up, pdg + 1, data%model)
    call flavor_init (flv_wm, - W_BOSON, data%model)
    m_out = flavor_get_mass (flv_up)
    call sf_int%base_init (mask, [data%mass**2], [m_out**2], &
        [data%mW**2], hel_lock = hel_lock)
    sf_int%data => data
    call quantum_numbers_init (qn_wm, flv_wm)

```

```

call polarization_init_generic (pol, data%flv)
call quantum_numbers_init (qn_fc(1), &
    flv = data%flv, col = color_from_flavor (data%flv, 1))
call quantum_numbers_init (qn_fc_fin(1), &
    flv = flv_up, col = color_from_flavor (flv_up))
call state_iterator_init (it_hel, pol%state)
do while (state_iterator_is_valid (it_hel))
    qn_hel = state_iterator_get_quantum_numbers (it_hel)
    qn = qn_hel(1) .merge. qn_fc(1)
    qn_up = qn_hel(1) .merge. qn_fc_fin(1)
    call interaction_add_state &
        (sf_int%interaction_t, (/ qn, qn_up, qn_wm /))
    call state_iterator_advance (it_hel)
end do
call polarization_final (pol)
end if
case default
    call msg_fatal ("EWA initialization failed: wrong particle type.")
end select
call interaction_freeze (sf_int%interaction_t)
!!! JRR: WK please check
!!! is that the right place to set the keep_ flags
if (data%keep_momentum) then
    if (data%keep_energy) then
        call msg_fatal ("EWA: momentum and energy" // &
            "cannot be simultaneously conserved.")
    else
        sf_int%on_shell_mode = KEEP_MOMENTUM
    end if
else
    if (data%keep_energy) then
        sf_int%on_shell_mode = KEEP_ENERGY
    end if
end if
call sf_int%set_incoming ([1])
call sf_int%set_radiated ([2])
call sf_int%set_outgoing ([3])
sf_int%status = SF_INITIAL
end select
end subroutine ewa_init

```

#### 10.6.4 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, the exponential mapping for the  $1/x$  singularity discussed above is applied.

```

<SF ewa: ewa: TBP>+≡
    procedure :: complete_kinematics => ewa_complete_kinematics
<SF ewa: procedures>+≡
    subroutine ewa_complete_kinematics (sf_int, x, f, r, map)
        !!! JRR: WK please check

```



```

class(ewa_t), intent(inout) :: sf_int
real(default), dimension(:), intent(out) :: x
real(default), intent(out) :: f
real(default), dimension(:), intent(in) :: r
logical, intent(in) :: map
real(default) :: xb1, x0, x1, lx0, lx1, lx
if (sf_int%data%keep_momentum .or. sf_int%data%keep_energy) then
  select case (sf_int%data%id)
    case (23)
      x0 = max (sf_int%data%x_min, sf_int%data%mz/sf_int%q)
    case (24)
      x0 = max (sf_int%data%x_min, sf_int%data%mw/sf_int%q)
    end select
else
  x0 = sf_int%data%x_min
end if
x1 = sf_int%data%x_max
if ( x0 >= x1) then
  f = 0
  sf_int%status = SF_FAILED_KINEMATICS
  return
end if
if (map) then
  lx0 = log (x0)
  lx1 = log (x1)
  lx = lx1 * r(1) + lx0 * (1 - r(1))
  x(1) = exp(lx)
  !!! JRR: WK please check, factor x would be part of Jacobian
  !!! but cancels 1/x in structure function. I do the cancellation
  !!! explicitly, depending on data%map
  f = lx1 - lx0      !!! x(1) * (lx1 - lx0)
else
  x(1) = r(1)
  f = 1
end if
xb1 = 1 - x(1)
if (size(x) == 3) x(2:3) = r(2:3)
!!! JRR: WK please check
!!! in kinematics we have to set the map flag in order
!!! to be available in apply
sf_int%data%map = map
call sf_int%split_momentum (x, xb1)
select case (sf_int%status)
case (SF_DONE_KINEMATICS)
  sf_int%x = x(1)
case (SF_FAILED_KINEMATICS)
  sf_int%x = 0
  f = 0
end select
end subroutine ewa_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the

same formula as for normal kinematics.

*<SF ewa: ewa: TBP>+≡*

```
procedure :: inverse_kinematics => ewa_inverse_kinematics
```

*<SF ewa: procedures>+≡*

```
subroutine ewa_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
  !!! JRR: WK please check
  class(ewa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1, x0, x1, lx0, lx1, lx
  logical :: set_mom
  set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
  if (sf_int%data%keep_momentum .or. sf_int%data%keep_energy) then
    select case (sf_int%data%id)
      case (23)
        x0 = max (sf_int%data%x_min, sf_int%data%mz/sf_int%q)
      case (24)
        x0 = max (sf_int%data%x_min, sf_int%data%mw/sf_int%q)
    end select
  else
    x0 = sf_int%data%x_min
  end if
  x1 = sf_int%data%x_max
  if (map) then
    lx0 = log (x0)
    lx1 = log (x1)
    r(1) = (log(x(1)) - lx0) / (lx1 - lx0)
    !!! JRR: WK please check, factor x would be part of Jacobian
    !!! but cancels 1/x in structure function. I do the cancellation
    !!! explicitly, depending on data%map
    f = lx1 - lx0    !!! x(1) * (lx1 - lx0)
  else
    r(1) = x(1)
    f = 1
  end if
  xb1 = 1 - x(1)
  if (size(r) == 3) r(2:3) = x(2:3)
  if (set_mom) then
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
      case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
  end if
end subroutine ewa_inverse_kinematics
```

### 10.6.5 EWA structure function

The EWA structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the  $x$  value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number  $r$  to generate the momentum fraction  $x$  and the function value  $f$ . For numerical stability reasons, we also output  $xb$ , which is  $\bar{x} = 1 - x$ . The mapping ensures that  $f = 1$  at leading order, and the higher-order corrections are well-behaved.

(EWA: procedures)≡

```
!! !! elemental subroutine strfun (f, fp, fm, fL, x, xb, r, E, data, no_map)
!! !!   real(default), intent(out) :: f, fm, fp, fL
!! !!   real(default) :: fsum
!! !!   real(default), intent(out) :: x, xb
!! !!   real(default), intent(in) :: r, E
!! !!   type(ewa_data_t), intent(in) :: data
!! !!   logical, intent(in) :: no_map
!! !!   real(default) :: x0, x1
!! !!   real(default) :: rb, lx0, lx1, lx, d, den
!! !!   real(default) :: c1, c2, pt2
!! !!   real(default) :: costhw, sinthw
!! !!   real(default) :: cv, ca, q, t3
!! !!   if (data%keep_momentum .or. data%keep_energy) then
!! !!     select case (data%id)
!! !!     case (23)
!! !!       x0 = max (data%x_min, data%mz/E)
!! !!     case (24)
!! !!       x0 = max (data%x_min, data%mw/E)
!! !!     end select
!! !!   else
!! !!     x0 = data%x_min
!! !!   end if
!! !!   x1 = data%x_max
!! !!   if (x0 >= x1) then
!! !!     f = 0
!! !!     fp = 0
!! !!     fm = 0
!! !!     fL = 0
!! !!     return
!! !!   end if
!! !!   lx0 = log (x0)
!! !!   lx1 = log (x1)
!! !!   lx = lx1 * r + lx0 * (1-r)
!! !!   x = exp(lx)
!! !!   xb = 1 - x
!! !!   f = data%coeff * (lx1 - lx0)
!! !!   pt2 = min ((data%pt_max)**2, (xb * data%sqrts / 2)**2)
!! !!   select case (data%id)
!! !!   case (23)
!! !!     !!! Z boson structure function
!! !!     c1 = log (1 + pt2 / (xb * (data%mZ)**2))
!! !!     c2 = 1 / (1 + (xb * (data%mZ)**2) / pt2)
```

```

!! !!      if (flavor_is_antiparticle (data%flv)) then
!! !!          q = flavor_get_charge (data%flv)
!! !!          t3 = flavor_get_isospin (data%flv)
!! !!      else
!! !!          q = - flavor_get_charge (data%flv)
!! !!          t3 = - flavor_get_isospin (data%flv)
!! !!      end if
!! !!      cv = data%cv * (t3 - 2._default * q * data%sinthw**2) / data%costhw
!! !!      ca = data%ca * t3 / data%costhw
!! !!      if (flavor_is_antiparticle (data%flv)) ca = - ca
!! !!      fm = ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
!! !!      fp = ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
!! !!      fL = (cv**2 + ca**2) * 2 * xb * c2
!! !!      fsum = fp + fm + fL
!! !!      f = f * fsum
!! !!      if (fsum /= 0) then
!! !!          fp = fp / fsum
!! !!          fm = fm / fsum
!! !!          fL = fL / fsum
!! !!      end if
!! !!      case (24)
!! !!          !!! W boson structure function
!! !!          c1 = log (1 + pt2 / (xb * (data%mw)**2))
!! !!          c2 = 1 / (1 + (xb * (data%mw)**2) / pt2)
!! !!          cv = data%cv / sqrt(2._default)
!! !!          ca = data%ca / sqrt(2._default)
!! !!          if (flavor_is_antiparticle (data%flv)) ca = - ca
!! !!          fm = ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
!! !!          fp = ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
!! !!          fL = (cv**2 + ca**2) * 2 * xb * c2
!! !!          fsum = fp + fm + fL
!! !!          f = f * fsum
!! !!          if (fsum /= 0) then
!! !!              fp = fp / fsum
!! !!              fm = fm / fsum
!! !!              fL = fL / fsum
!! !!          end if
!! !!      end select
!! !!  end subroutine strfun

```

### 10.6.6 EWA application

For EWA, we can compute kinematics and function value in a single step. This function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for  $x$ , one for  $Q^2$ , and two for the polar and azimuthal angles. Alternatively, we can skip  $p_T$  generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated  $x$  values are identical.

```

⟨SF ewa: ewa: TBP⟩+≡
  procedure :: apply => ewa_apply

```

```

<SF ewa: procedures>+≡
subroutine ewa_apply (sf_int, scale) !! r, ewa_data, no_map)
  class(ewa_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: x, xb, pt2, c1, c2, q, t3
  real(default) :: cv, ca
  real(default) :: f, fm, fp, fL
  associate (data => sf_int%data)
    x = sf_int%x
    xb = 1 - x
    fm = 0
    fp = 0
    fL = 0
    !!! JRR: WK please check: should be correct. The structure
    !!! function depends on mapped variables in the mapped case
    !!! and the true r values (constrained to interval [xmin, xmax])
    !!! for the unmapped case. Jacobian is taken into account in
    !!! complete_kinematics. Is this true also for the qminsq bound
    !!! or do we have to use the unmapped x values there?
    !!! Furthermore: in W 2.1.1 fm, fp, fL have not been used...
    pt2 = min ((data%pt_max)**2, (xb * data%sqrts / 2)**2)
    select case (data%id)
    case (23)
      !!! Z boson structure function
      c1 = log (1 + pt2 / (xb * (data%mZ)**2))
      c2 = 1 / (1 + (xb * (data%mZ)**2) / pt2)
      if (flavor_is_antiparticle (data%flv)) then
        q = flavor_get_charge (data%flv)
        t3 = flavor_get_isospin (data%flv)
      else
        q = - flavor_get_charge (data%flv)
        t3 = - flavor_get_isospin (data%flv)
      end if
      cv = data%cv * (t3 - 2._default * q * data%sinhw**2) / data%coshw
      ca = data%ca * t3 / data%coshw
      if (flavor_is_antiparticle (data%flv)) ca = - ca
      fm = data%coeff * &
        ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
      fp = data%coeff * &
        ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
      fL = data%coeff * &
        (cv**2 + ca**2) * 2 * xb * c2
      if (.not. data%map) then
        fm = fm / x
        fp = fp / x
        fL = fL / x
      end if
      f = fp + fm + fL
      if (f /= 0) then
        fp = fp / f
        fm = fm / f
        fL = fL / f
      end if
    case (24)

```

```

!!! W boson structure function
c1 = log (1 + pt2 / (xb * (data%mw)**2))
c2 = 1 / (1 + (xb * (data%mw)**2) / pt2)
cv = data%cv / sqrt(2._default)
ca = data%ca / sqrt(2._default)
if (flavor_is_antiparticle (data%flv)) ca = - ca
fm = data%coeff * &
      ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
fp = data%coeff * &
      ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
fL = data%coeff * &
      (cv**2 + ca**2) * 2 * xb * c2
if (.not. data%map) then
  fm = fm / x
  fp = fp / x
  fL = fL / x
end if
f = fp + fm + fL
if (f /= 0) then
  fp = fp / f
  fm = fm / f
  fL = fL / f
end if
end select
end associate
call interaction_set_matrix_element &
      (sf_int%interaction_t, cmplx (f, kind=default))
sf_int%status = SF_EVALUATED
end subroutine ewa_apply

```

### 10.6.7 Unit tests

```

<SF ewa: public>+≡
  public :: sf_ewa_test

<SF ewa: tests>≡
  subroutine sf_ewa_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    !!! JRR: WK please check
    !!! there are probably more tests necessary. e.g. to see whether
    !!! integrals with and without mapping are the same, right?
  <SF ewa: execute tests>
  end subroutine sf_ewa_test

```

### Test structure function data

Construct and display a test structure function data object.

```

<SF ewa: execute tests>≡
  call test (sf_ewa_1, "sf_ewa_1", &
    "structure function configuration", &
    u, results)

```

```

<SF ewa: tests> +=
subroutine sf_ewa_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_ewa_1"
  write (u, "(A)")  "* Purpose: initialize and display &
    &test structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("SM"), &
    var_str ("SM.mdl"), os_data, model)
  call flavor_init (flv, 2, model)

  allocate (ewa_data_t :: data)
  call data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize for Z boson"
  write (u, "(A)")

  select type (data)
  type is (ewa_data_t)
    call data%init (model, flv, 0.01_default, 10._default, &
      500._default, 5000._default, .false., .false., 5._default)
    call data%set_id (23)
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle codes:"
  write (u, "(2x,99(1x,I0))")  data%get_pdg_out ()

  write (u, "(A)")
  write (u, "(A)")  "* Initialize for W boson"
  write (u, "(A)")

  select type (data)
  type is (ewa_data_t)
    call data%init (model, flv, 0.01_default, 10._default, &
      500._default, 5000._default, .false., .false., 5._default)
    call data%set_id (24)
  end select

```

```

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)") "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))") data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_ewa_1"

end subroutine sf_ewa_1

```

### Test and probe structure function

Construct and display a structure function object based on the EWA structure function.

```

<SF ewa: execute tests>+≡
call test (sf_ewa_2, "sf_ewa_2", &
  "structure function instance", &
  u, results)

<SF ewa: tests>+≡
subroutine sf_ewa_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, x
  real(default) :: f, s

  write (u, "(A)")  "* Test output: sf_ewa_2"
  write (u, "(A)")  "* Purpose: initialize and fill &
    &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("SM"), &
    var_str ("SM.mdl"), os_data, model)
  call flavor_init (flv, 2, model)

  call reset_interaction_counter ()

```



```

allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
    call data%init (model, flv, 0.01_default, 10._default, &
        500._default, 5000._default, .false., .false., 5._default)
    call data%set_id (24)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, no EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0.5_default
select type (sf_int)
type is (ewa_t)
    sf_int%q = 1000._default
end select
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)

```

```

call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F12.9))") "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate EWA structure function"
write (u, "(A)")

select type (sf_int)
type is (ewa_t)
    sf_int%q = 1000._default
end select
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

call sf_int%final ()
deallocate (sf_int)
deallocate (x)
deallocate (r)
deallocate (data)

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
    call data%init (model, flv, 0.01_default, 10._default, &
        500._default, 5000._default, .false., .true., 5._default)
    call data%set_id (24)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

```

```

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.25, with EWA mapping, "
write (u, "(A)")  "          non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
select type (sf_int)
type is (ewa_t)
    sf_int%q = 1000._default
end select
call sf_int%complete_kinematics (x, f, r, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
select type (sf_int)
type is (ewa_t)
    sf_int%q = 1000._default
end select
call sf_int%inverse_kinematics (x, f, r, map=.true.)

write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Evaluate EWA structure function"
write (u, "(A)")

select type (sf_int)
type is (ewa_t)
    sf_int%q = 1000._default
end select
call sf_int%complete_kinematics (x, f, r, map=.true.)

```

```

call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%apply (scale = 1000._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_ewa_2"

end subroutine sf_ewa_2

```

## 10.7 Lepton collider beamstrahlung: CIRCE1

```
<sf_circe1.f90>≡  
<File header>  
  
module sf_circe1  
  
  <Use kinds>  
  <Use strings>  
  <Use file utils>  
  use limits, only: CIRCE1_EPSILON !NODEP!  
  use diagnostics !NODEP!  
  use tao_random_numbers !NODEP!  
  use lorentz !NODEP!  
  use models  
  use flavors  
  use colors  
  use quantum_numbers  
  use state_matrices  
  use polarizations  
  use interactions  
  use sf_aux  
  use sf_base  
  use circe1 !NODEP!  
  
  <Standard module head>  
  
  <SF circe1: public>  
  
  <SF circe1: parameters>  
  
  <SF circe1: types>  
  
  <SF circe1: variables>  
  
  contains  
  
  <SF circe1: procedures>  
  
end module sf_circe1
```

### 10.7.1 Physics

Beamstrahlung is applied before ISR. The CIRCE1 implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously). Nevertheless it is factorized:

The functional form in the CIRCE1 parameterization is defined for electrons or photons

$$f(x) = \alpha x^\beta (1-x)^\gamma \quad (10.64)$$

for  $x < 1 - \epsilon$  (resp.  $x > \epsilon$  in the photon case). In the remaining interval, the standard form is zero, with a delta singularity at  $x = 1$  (resp.  $x = 0$ ). Equivalently, the delta part may be distributed uniformly among this interval.

This latter form is implemented in the `kirke` version of the `CIRCE1` subroutines, and is used here.

The parameter  $\epsilon$  is hardcoded in `CIRCE1` (set equal to  $10^{-6}$ ). Therefore, it cannot be changed in the namelist.

```
<SF circe1: public parameters>≡
  real(kind=default), parameter, public :: KIREPS = 1E-6_default
```

The other parameters are the parameterization version and revision number, the accelerator type, and the  $\sqrt{s}$  value used by `CIRCE1`. The chattiness can also be set (?).

Since the energy is distributed in a narrow region around unity (for electrons) or zero (for photons), it is advantageous to map the interval first. The mapping is controlled by the powers `beta` and `gamma` which are taken from the `CIRCE1` internal data structure.

The  $\sqrt{s}$  value, if not explicitly set, is taken from the process data. Note that interpolating  $\sqrt{s}$  is not recommended; one should rather choose one of the distinct values known to `CIRCE1`.

### 10.7.2 The `CIRCE1` data block

The `CIRCE1` parameters are: The incoming flavors, the flags whether the photon or the lepton is the parton in the hard interaction, the flags for the generation mode (generator/mapping/no mapping), the mapping parameters  $\beta$  and  $\gamma$ ,  $\sqrt{s}$  and several steering parameters: `ver`, `rev`, `acc`, `chat`.

In generator mode, the  $x$  values are actually discarded and a random number generator is used instead.

```
<SF circe1: public>≡
  public :: circe1_data_t

<SF circe1: types>≡
  type, extends (sf_data_t) :: circe1_data_t
    private
    type(model_t), pointer :: model => null ()
    type(flavor_t), dimension(2) :: flv_in
    integer, dimension(2) :: pdg_in
    real(default), dimension(2) :: m_in = 0
    logical, dimension(2) :: photon = .false.
    logical :: generate = .true.
    type(tao_random_state), pointer :: rng => null ()
    logical :: map = .true.
    real(default), dimension(2) :: beta = 0
    real(default), dimension(2) :: gamma = 0
    real(default) :: sqrts = 0
    integer :: ver = 0
    integer :: rev = 0
    integer :: acc = 0
    integer :: chat = 0
    integer :: error = NONE
  contains
    <SF circe1: circe1 data: TBP>
  end type circe1_data_t
```

Error codes

```

<SF circe1: parameters>≡
  integer, parameter :: NONE = 0
  integer, parameter :: FLV_INAPPLICABLE = 1

<SF circe1: circe1 data: TBP>≡
  procedure :: init => circe1_data_init

<SF circe1: procedures>≡
  subroutine circe1_data_init &
    (data, model, flv, sqrts, out_photon, generate, rng, map, &
     ver, rev, acc, chat)
    class(circe1_data_t), intent(out) :: data
    type(model_t), intent(in), target :: model
    type(flavor_t), dimension(2), intent(in) :: flv
    real(default), intent(in) :: sqrts
    logical, dimension(2), intent(in) :: out_photon
    logical, intent(in) :: generate, map
    type(tao_random_state), intent(in), target :: rng
    integer, intent(in) :: ver, rev, acc, chat
    data%model => model
    data%flv_in = flv
    data%pdg_in = flavor_get_pdg (data%flv_in)
    if (any (abs (data%pdg_in) /= ELECTRON)) data%error = FLV_INAPPLICABLE
    if (data%pdg_in(1) /= - data%pdg_in(2)) data%error = FLV_INAPPLICABLE
    data%m_in = flavor_get_mass (data%flv_in)
    data%sqrts = sqrts
    data%photon = out_photon
    data%generate = generate
    data%rng => rng
    data%map = map
    data%ver = ver
    data%rev = rev
    data%acc = acc
    data%chat = chat
    select case (char (model_get_name (data%model)))
    case ("QCD", "Test")
      call msg_fatal ("CIRCE1 structure function not available for model " &
        // char (model_get_name (data%model)) // ".")
    end select
    call circes (0.d0, 0.d0, dble (data%sqrts), &
      data%acc, data%ver, data%rev, data%chat)
    call circe1_get_parameters (data%photon, data%beta, data%gamma)
  end subroutine circe1_data_init

```

These procedures access the CIRCE internal state directly (!).

```

<SF circe1: procedures>+≡
  elemental subroutine circe1_get_parameters (photon, beta, gamma)
    logical, intent(in) :: photon
    real(default), intent(out) :: beta, gamma
    if (photon) then
      beta = circe1_params%a1(6)
      gamma = circe1_params%a1(5)
    else

```

```

        beta = circe1_params%a1(2)
        gamma = circe1_params%a1(3)
    end if
end subroutine circe1_get_parameters

```

Handle error conditions.

```

<CIRCE1: public>≡
    public :: circe1_data_check

<CIRCE1: procedures>≡
    subroutine circe1_data_check (data)
        type(circe1_data_t), intent(in) :: data
        select case (data%error)
        case (FLV_INAPPLICABLE)
            call msg_fatal ("CIRCE1: applicable only for incoming " &
                // "electron or positron")
        end select
    end subroutine circe1_data_check

```

Output

```

<SF circe1: circe1 data: TBP>+≡
    procedure :: write => circe1_data_write

<SF circe1: procedures>+≡
    subroutine circe1_data_write (data, unit, verbose) !, md5)
        class(circe1_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        !!! JRR: WK please check: still needed?
    !    logical, intent(in), optional :: md5
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "CIRCE1 data:"
        write (u, "(3x,A,A,A,A)") " prt_in = ", &
            char (flavor_get_name (data%flv_in(1))), &
            ", ", char (flavor_get_name (data%flv_in(2)))
        write (u, "(3x,A,L1)") " photon = ", data%photon
        write (u, "(3x,A,L1)") " generate = ", data%generate
        write (u, "(3x,A,L1)") " map = ", data%map
        write (u, "(1x,A,ES19.12)") " beta = ", data%beta
        write (u, "(1x,A,ES19.12)") " gamma = ", data%gamma
        write (u, "(1x,A,ES19.12)") " m_in = ", data%m_in
        write (u, "(1x,A,ES19.12)") " sqrts = ", data%sqrts
        write (u, "(3x,I3)") " ver = ", data%ver
        write (u, "(3x,I3)") " rev = ", data%rev
        write (u, "(3x,I3)") " acc = ", data%acc
        write (u, "(3x,A,L1)") " chat = ", data%chat
    end subroutine circe1_data_write

```

The number of parameters is two, collinear splitting for the two beams.

```

<SF circe1: circe1 data: TBP>+≡
    procedure :: get_n_par => circe1_data_get_n_par

```



```

<SF circe1: procedures>+≡
  function circe1_data_get_n_par (data) result (n)
    class(circe1_data_t), intent(in) :: data
    integer :: n
    n = 2
  end function circe1_data_get_n_par

```

Return the outgoing particles PDG codes. This is either the incoming particle (if a photon is radiated), or the photon if that is the particle of the hard interaction. The latter is determined via the photon flag. There are two entries for the two beams (abuse?).

```

<SF circe1: circe1 data: TBP>+≡
  procedure :: get_pdg_out => circe1_data_get_pdg_out

<SF circe1: procedures>+≡
  !!! JRR: WK please check
  function circe1_data_get_pdg_out (data) result (pdg_out)
    class(circe1_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    integer :: i, n
    n = 2
    allocate (pdg_out (n))
    do i = 1, n
      if (data%photon(i)) then
        pdg_out(i) = PHOTON
      else
        pdg_out(i) = data%pdg_in(i)
      end if
    end do
  end function circe1_data_get_pdg_out

```

Allocate the interaction record.

```

<SF circe1: circe1 data: TBP>+≡
  procedure :: allocate_sf_int => circe1_data_allocate_sf_int

<SF circe1: procedures>+≡
  subroutine circe1_data_allocate_sf_int (data, sf_int)
    class(circe1_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (circe1_t:: sf_int)
  end subroutine circe1_data_allocate_sf_int

```

### 10.7.3 The CIRCE1 object

This is a  $2 \rightarrow 4$  interaction, where, depending on the parameters, any two of the four outgoing particles are connected to the hard interactions, the others are radiated. Knowing that all particles are colorless, we do not have to deal with color.

The flavors are sorted such that the first two particles are the incoming leptons, the next two are the radiated particles, and the last two are the partons initiating the hard interaction.

CIRCE1 does not support polarized beams explicitly. For simplicity, we nevertheless carry beam polarization through to the outgoing electrons and make the photons unpolarized.

```

<SF circe1: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: circe1_t
    type(circe1_data_t), pointer :: data => null ()
    real(default), dimension(2) :: x = 0
    real(default), dimension(2) :: q = 0
  contains
    <SF circe1: circe1: TBP>
  end type circe1_t

```

Type string: has to here, but there is no string variable on which CIRCE1 depends. Hence, a dummy routine.

```

<SF circe1: circe1: TBP>≡
  procedure :: type_string => circe1_type_string

<SF circe1: procedures>+≡
  function circe1_type_string (object) result (string)
    class(circe1_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "CIRCE1: beamstrahlungs spectrum"
    else
      string = "CIRCE1: [undefined]"
    end if
  end function circe1_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF circe1: circe1: TBP>+≡
  procedure :: write => circe1_write

<SF circe1: procedures>+≡
  subroutine circe1_write (object, unit)
    !!! JRR: WK please check
    !!! Guess these variables do not exist for CIRCE1 (?)
    class(circe1_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)") "SF parameters:"
        write (u, "(3x,A,ES17.10)") "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
          write (u, "(3x,A,ES17.10)") "Q =", object%q
        end if
      end if
      call object%base_write (u)
    else
      write (u, "(1x,A)") "CIRCE1 data: [undefined]"
    end if
  end subroutine circe1_write

```

```

    end if
end subroutine circe1_write

```

#### 10.7.4 Kinematics

Set kinematics. If *map* is unset, the *r* and *x* values coincide, and the Jacobian  $f(r)$  is trivial.

If *map* is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF circe1: circe1: TBP>+≡
  procedure :: complete_kinematics => circe1_complete_kinematics

<SF circe1: procedures>+≡
  subroutine circe1_complete_kinematics (sf_int, x, f, r, map)
    !!! JRR: WK please check
    class(circe1_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    logical, intent(in) :: map
    real(default), dimension(2) :: xb1
    if (map) then
      call msg_fatal ("CIRCE1: map flag not supported")
    else
      x = r
      f = 1
    end if
    xb1 = 1 - x
    call sf_int%split_momenta (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
      sf_int%x = x(1:2)
    case (SF_FAILED_KINEMATICS)
      sf_int%x = 0
      f = 0
    end select
  end subroutine circe1_complete_kinematics

```

Compute inverse kinematics. Here, we start with the *x* array and compute the “input” *r* values and the Jacobian *f*. After this, we can set momenta by the same formula as for normal kinematics.

```

<SF circe1: circe1: TBP>+≡
  procedure :: inverse_kinematics => circe1_inverse_kinematics

<SF circe1: procedures>+≡
  subroutine circe1_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
    !!! JRR: WK please check
    !!! This cannot be correct, as the CIRCE1 structure function has
    !!! twice the variables (2->4 instead of 1->2 splitting)
    class(circe1_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f

```

```

real(default), dimension(:), intent(out) :: r
logical, intent(in) :: map
logical, intent(in), optional :: set_momenta
real(default), dimension(2) :: xb1
logical :: set_mom
set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
if (map) then
    call msg_fatal ("CIRCE1: map flag not supported")
else
    r = x
    f = 1
end if
if (set_mom) then
    xb1 = 1 - x
    call sf_int%split_momenta (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
        sf_int%x = x(1:2)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end if
end subroutine circe1_inverse_kinematics

```

$\langle SF \text{ circe1: circe1: TBP} \rangle + \equiv$

```

procedure :: init => circe1_init

```

$\langle SF \text{ circe1: procedures} \rangle + \equiv$

```

subroutine circe1_init (sf_int, data)
    class(circe1_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    logical, dimension(6) :: mask_h
    type(quantum_numbers_mask_t), dimension(6) :: mask
    integer, dimension(6) :: hel_lock
    type(polarization_t) :: pol1, pol2
    type(quantum_numbers_t), dimension(1) :: qn_fc1, qn_hel1, qn_fc2, qn_hel2
    type(flavor_t) :: flv_photon
    real(default), dimension(2) :: mi2, mr2, mo2
    type(quantum_numbers_t) :: qn_photon, qn1, qn2
    type(quantum_numbers_t), dimension(6) :: qn
    type(state_iterator_t) :: it_hel1, it_hel2
    hel_lock = 0
    mask_h = .false.
    select type (data)
    type is (circe1_data_t)
        mi2 = data%m_in**2
        if (data%photon(1)) then
            hel_lock(1) = 3; hel_lock(3) = 1; mask_h(5) = .true.
            mr2(1) = mi2(1)
            mo2(1) = 0._default
        else
            hel_lock(1) = 5; hel_lock(5) = 1; mask_h(3) = .true.
            mr2(1) = 0._default

```

```

        mo2(1) = mi2(1)
    end if
    if (data%photon(2)) then
        hel_lock(2) = 4; hel_lock(4) = 2; mask_h(6) = .true.
        mr2(2) = mi2(2)
        mo2(2) = 0._default
    else
        hel_lock(2) = 6; hel_lock(6) = 2; mask_h(4) = .true.
        mr2(2) = 0._default
        mo2(2) = mi2(2)
    end if
    mask = new_quantum_numbers_mask (.false., .false., mask_h)
    call sf_int%base_init (mask, mi2, mr2, mo2, &
        hel_lock = hel_lock)
    sf_int%data => data
    call flavor_init (flv_photon, PHOTON, data%model)
    call quantum_numbers_init (qn_photon, flv_photon)
    call polarization_init_generic (pol1, data%flv_in(1))
    call quantum_numbers_init (qn_fc1(1), flv = data%flv_in(1))
    call polarization_init_generic (pol2, data%flv_in(2))
    call quantum_numbers_init (qn_fc2(1), flv = data%flv_in(2))
    call state_iterator_init (it_hel1, pol1%state)
    do while (state_iterator_is_valid (it_hel1))
        qn_hel1 = state_iterator_get_quantum_numbers (it_hel1)
        qn1 = qn_hel1(1) .merge. qn_fc1(1)
        qn(1) = qn1
        if (data%photon(1)) then
            qn(3) = qn1; qn(5) = qn_photon
        else
            qn(3) = qn_photon; qn(5) = qn1
        end if
        call state_iterator_init (it_hel2, pol2%state)
        do while (state_iterator_is_valid (it_hel2))
            qn_hel2 = state_iterator_get_quantum_numbers (it_hel2)
            qn2 = qn_hel2(1) .merge. qn_fc2(1)
            qn(2) = qn2
            if (data%photon(2)) then
                qn(4) = qn2; qn(6) = qn_photon
            else
                qn(4) = qn_photon; qn(6) = qn2
            end if
            call interaction_add_state (sf_int%interaction_t, qn)
            call state_iterator_advance (it_hel2)
        end do
        call state_iterator_advance (it_hel1)
    end do
    call polarization_final (pol1)
    call polarization_final (pol2)
    call interaction_freeze (sf_int%interaction_t)
    !!! JRR: WK please check
    !!! Is this correct if the photon is the radiating particle??
    call sf_int%set_incoming ([1,2])
    call sf_int%set_radiated ([3,4])
    call sf_int%set_outgoing ([5,6])

```

```

        sf_int%status = SF_INITIAL
    end select
end subroutine circe1_init

```

### 10.7.5 CIRCE1 structure function

Compute the CIRCE1 structure function: either using the generator or the structure function value. In the latter case, we may employ a mapping that depends on the  $\beta, \gamma$  parameters.

The generator mode is the default and by far more useful, since there is virtually no loss in accuracy or reweighting efficiency for a process with CIRCE1 and without. The formal problem with this approach is the indeterministic integration, which is presumably harmless since the corresponding integration dimensions factorize in the VEGAS approach.

If `no_map` is set, we switch off generator and mapping.

```

<Limits: public parameters>+=
    double precision, parameter, public :: CIRCE1_EPSILON = 1d-6

<CIRCE1: procedures>+=
    subroutine strfun (f, x, xb, r, circe1_data, no_map)
        real(default), intent(out) :: f
        real(default), dimension(2), intent(out) :: x, xb
        real(default), dimension(2), intent(in) :: r
        type(circe1_data_t), intent(in) :: circe1_data
        logical, intent(in) :: no_map
        real(kind=default), parameter :: eps = CIRCE1_EPSILON
        real(default), dimension(2) :: fi
        if (circe1_data%generate .and. .not. no_map) then
            call circe_generate &
                (x, circe1_data%rng, circe1_data%pdg_in < 0, circe1_data%photon)
            xb = 1 - x
            f = 1
        else if (circe1_data%map .and. .not. no_map) then
            call circe_map_p (fi, x, xb, r, eps, &
                circe1_data%beta, circe1_data%gamma, circe1_data%photon)
            f = product (fi) * strfun_circe1 (x, circe1_data%pdg_in)
        else
            x = r
            xb = 1 - r
            f = strfun_circe1 (x, circe1_data%pdg_in)
        end if
    end subroutine strfun

```

Call the CIRCE1 generator. The integration variables are not used in this case. Instead, we call a random-number generator.

The CIRCE1 generator calls need the generator as argument, but not the generator state. Thus, we temporarily assign a pointer to the state to a module variable which is accessed by the generator.

```

<CIRCE1: variables>=
    type(tao_random_state), pointer :: rng_tmp => null ()

```

```

<CIRCE1: procedures>+≡
  subroutine rn_sub (r)
    double precision, intent(out) :: r
    real(default) :: x
    call tao_random_number (rng_tmp, x)
    r = x
  end subroutine rn_sub

<CIRCE1: procedures>+≡
  subroutine circe_generate (x, rng, anti, photon)
    logical, dimension(2), intent(in) :: anti
    logical, dimension(2), intent(in) :: photon
    real(default), dimension(2), intent(out) :: x
    real(double), dimension(2) :: xdum
    type( tao_random_state ), intent(in), target :: rng
    rng_tmp => rng
    if (all (photon)) then
      call gircgg (xdum(1), xdum(2), rn_sub)
    else if (photon(2)) then
      call girceg (xdum(1), xdum(2), rn_sub)
    else if (photon(1)) then
      call girceg (xdum(2), xdum(1), rn_sub)
    else if (.not. anti(1) .and. anti(2)) then
      call gircee (xdum(1), xdum(2), rn_sub)
    else if (anti(1) .and. .not. anti(2)) then
      call gircee (xdum(2), xdum(1), rn_sub)
    else
      call msg_bug ("CIRCE1: impossible flavor assignment")
    end if
    x = xdum
    rng_tmp => null ()
  end subroutine circe_generate

```

Apply the appropriate mapping, exchanging  $x$  vs.  $1 - x$  depending on the particle. This can be done for both beams simultaneously, since the parameterization factorizes.

The CIRCE1 structure function has the form of a  $\beta$  distribution

$$f(x) = \begin{cases} \alpha x^\beta (1-x)^\gamma & \text{for } 0 < x < x_0 \\ \alpha x_0^\beta (1-x_0)^\gamma & \text{for } x_0 < x < 1 \end{cases} \quad (10.65)$$

in the electron/positron case. In the photon case, this form holds for  $1 - x$ . Instead of mapping this function in one step, we consider the two factors separately, such that the mapping  $y(x)$  satisfies either

$$\frac{dy}{dx} = \begin{cases} ax^\beta & \text{for } 0 < x < x_0 \\ ax_0^\beta & \text{for } x_0 < x < 1 \end{cases} \quad (10.66)$$

or

$$\frac{dy}{dx} = \begin{cases} a(1-x)^\gamma & \text{for } 0 < x < x_0 \\ a(1-x_0)^\gamma & \text{for } x_0 < x < 1 \end{cases} \quad (10.67)$$

where, typically,  $\beta > 0$  and  $\gamma < 0$ .

Integrating this leads to

$$y = \begin{cases} a \frac{x^{1+\beta}}{a + \beta}, & x < x_0 \\ y_0 + a(x - x_0)x_0^\beta, & x > x_0 \end{cases} \quad (10.68)$$

resp.

$$y = \begin{cases} a \frac{1 - (1 - x)^{1+\gamma}}{1 + \gamma}, & x < x_0 \\ y_0 + ax_0^\gamma(x - x_0) & x > x_0 \end{cases} \quad (10.69)$$

We need the inverse mapping: The  $y$  value is taken from the random number set and has to be transformed into a  $x$  value which is fed into **CIRCE1**.

$$x = \begin{cases} \left( (1 + \beta) \frac{y}{a} \right)^{\frac{1}{1+\beta}} & y < y_0 \\ x_0 + \frac{y - y_0}{ax_0^\beta} & y > y_0 \end{cases} \quad (10.70)$$

or

$$x = \begin{cases} 1 - \left( 1 - (1 + \gamma) \frac{y}{a} \right)^{\frac{1}{1+\gamma}} & y < y_0 \\ x_0 + \frac{y - y_0}{a(1 - x_0)^\gamma} & y > y_0 \end{cases} \quad (10.71)$$

The boundary in  $y$  is given by

$$y_0 = a \frac{x_0^{1+\beta}}{1 + \beta} \quad \text{resp.} \quad y_0 = a \frac{1 - (1 - x_0)^{1+\gamma}}{1 + \gamma} \quad (10.72)$$

and the normalization is chosen such that  $y(1) = 1$

$$a = \left( \frac{x_0^{1+\beta}}{1 + \beta} + (1 - x_0)x_0^\beta \right)^{-1} \quad (10.73)$$

or

$$a = \left( \frac{1 - (1 - x_0)^{1+\gamma}}{1 + \gamma} + (1 - x_0)^{1+\gamma} \right)^{-1} \quad (10.74)$$

*(CIRCE1: procedures)* +=

```

elemental subroutine circe_map_p (f, x, xb, r, eps, beta, gamma, photon)
  real(default), intent(out) :: f, x, xb
  real(default), intent(in) :: r
  real(default), intent(in) :: eps, beta, gamma
  logical, intent(in) :: photon
  real(default) :: rb
  if (photon) then
    rb = 1 - r
    call circe_map_s (f, xb, rb, 1-eps, beta, gamma)
    x = 1 - xb
  else
    call circe_map_s (f, x, r, 1-eps, beta, gamma)
    xb = 1 - x
  end if
end subroutine circe_map_p

```



Apply the mapping. We map both powers by concatenating the mappings.

*(CIRCE1: procedures)*+≡

```

pure subroutine circe_map_s (f, x, z, x0, beta, gamma)
  real(default), intent(out) :: f, x
  real(default), intent(in) :: z, x0, beta, gamma
  real(default) :: y, ay, az, y0, z0
  call circe_set_const (ay, y0, x0, gamma, invert=.true.)
  call circe_set_const (az, z0, y0, beta, invert=.false.)
  y = circe_map_x (z, z0, y0, az, beta, invert=.false.)
  x = circe_map_x (y, y0, x0, ay, gamma, invert=.true.)
  f = 1 / circe_jacobian (x, x0, ay, gamma, invert=.true.) &
        / circe_jacobian (y, y0, az, beta, invert=.false.)
end subroutine circe_map_s

```

*(CIRCE1: procedures)*+≡

```

pure function circe_map_x (y, y0, x0, a, power, invert) result (x)
  real(kind=default), intent(in) :: y, y0, x0, a, power
  logical, intent(in) :: invert
  real(kind=default) :: x
  if (y<=0) then
    x = 0
  else if (y>1) then
    x = 1
  else if (invert) then
    if (y<y0) then
      x = 1 - (1 - (1+power)*y/a)**(1/(1+power))
    else
      x = x0 + (y-y0)/(a*(1-x0)**power)
    end if
  else
    if (y<y0) then
      x = ((1+power)*y/a)**(1/(1+power))
    else
      x = x0 + (y-y0)/(a*x0**power)
    end if
  end if
end function circe_map_x

```

Set constants for power mappings.

*(CIRCE1: procedures)*+≡

```

pure subroutine circe_set_const (a, y0, x0, power, invert)
  real(default), intent(out) :: a, y0
  real(default), intent(in) :: x0, power
  logical, intent(in) :: invert
  real(default) :: tmp
  if (invert) then
    tmp = (1 - (1-x0)**(1+power)) / (1+power)
    a = 1 / (tmp + (1-x0)**(1+power))
  else
    tmp = x0**(1+power) / (1+power)
    a = 1 / (tmp + (1-x0) * x0**power)
  end if
  y0 = a * tmp

```

```
end subroutine circe_set_const
```

The Jacobian factor for the power mapping:

*<CIRCE1: procedures>+≡*

```
pure function circe_jacobian (x, x0, a, power, invert) result (dy_dx)
  real(kind=default), intent(in) :: x, x0, a, power
  logical, intent(in) :: invert
  real(kind=default) :: dy_dx
  if (x<=0) then
    dy_dx = a
  else if (invert) then
    if (x<x0) then
      dy_dx = a * (1-x)**power
    else
      dy_dx = a * (1-x0)**power
    end if
  else
    if (x<x0) then
      dy_dx = a * x**power
    else
      dy_dx = a * x0**power
    end if
  end if
end function circe_jacobian
```

The actual CIRCE call. The parameters are already mapped, so we can directly call the structure function.

*<CIRCE1: procedures>+≡*

```
function strfun_circe1 (x, pdg) result (f)
  real(default) :: f
  real(default), dimension(2), intent(in) :: x
  integer, dimension(2), intent(in) :: pdg
  if (all (x /= 0)) then
    f = kirke (dble (x(1)), dble (x(2)), pdg(1), pdg(2))
  else
    f = 0
  end if
end function strfun_circe1
```

### 10.7.6 CIRCE1 application

CIRCE is applied for the two beams at once. We can safely assume that no structure functions are applied before this, so the incoming particles are on-shell electrons/positrons. There will however be at some point the lumi-linker implementations, which are used as an alternative to the CIRCE1 structure function.

*<SF circe1: circe1: TBP>+≡*

```
procedure :: apply => circe1_apply
```

```

<SF circe1: procedures>+=
subroutine circe1_apply (sf_int, scale) !!!, no_map)
!!! subroutine circe1_apply (sf_int, r, no_map)
!!! JRR: WK please check
!!! this somehow is different from the PDFs, so the types
!!! do not match. Guess we need two types of strfun,
!!! 1->2 and 2->4, nez-pas?
class(circe1_t), intent(inout) :: sf_int
!!! Dummy
real(default), intent(in) :: scale
!!!real(default), dimension(2), intent(in) :: r
!!!logical, intent(in) :: no_map
!!!type(vector4_t), dimension(2) :: k
!!!type(splitting_data_t), dimension(2) :: sd
!!!real(default), dimension(2) :: m_in, x, xb
real(default) :: f
!!!type(vector4_t), dimension(4) :: q
associate (data => sf_int%data)
!!! !!! JRR: WKK please check
!!! !!! No clue what to do here :(((
!!! k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
!!! k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
!!! m_in = data%m_in
!!! sd = new_splitting_data (k, m_in**2, m_in**2, m_in)
!!! call strfun (f, x, xb, r, circe1_data, no_map)
!!! call splitting_set_t_bounds (sd, x, xb)
!!! call splitting_set_collinear (sd)
!!! q((1, 3/)) = split_momentum (k(1), sd(1))
!!! q((2, 4/)) = split_momentum (k(2), sd(2))
!!! call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
end associate
call interaction_set_matrix_element &
(sf_int%interaction_t, cmplx (f, kind=default))
sf_int%status = SF_EVALUATED
end subroutine circe1_apply

```

## 10.8 Lepton collider beamstrahlung: Lumi Linker

```
<sf_lumi_linker.f90>≡  
  <File header>  
  
  module sf_lumi_linker  
  
    <Use kinds>  
    <Use strings>  
    <Use file utils>  
    use diagnostics !NODEP!  
    use tao_random_numbers !NODEP!  
    use lorentz !NODEP!  
    use models  
    use flavors  
    use colors  
    use quantum_numbers  
    use state_matrices  
    use polarizations  
    use interactions  
    use sf_aux  
    use sf_base  
  
    <Standard module head>  
  
    <SF lumi linker: public>  
  
    <SF lumi linker: parameters>  
  
    <SF lumi linker: types>  
  
    <SF lumi linker: variables>  
  
    contains  
  
    <SF lumi linker: procedures>  
  
  end module sf_lumi_linker
```

### 10.8.1 Physics

Lumi linker implementation by Tim Barklow. Dummy at the moment.

### 10.8.2 The Lumi Linker data block

```
<SF lumi linker: public>≡  
  public :: lumi_linker_data_t  
  
<SF lumi linker: types>≡  
  type, extends (sf_data_t) :: lumi_linker_data_t  
    private  
    type(model_t), pointer :: model => null ()  
    type(flavor_t), dimension(2) :: flv_in  
    integer, dimension(2) :: pdg_in
```

```

    real(default) :: sqrts
contains
    <SF lumi linker: lumi linker data: TBP>
end type lumi_linker_data_t

```

```

<SF lumi linker: lumi linker data: TBP>≡
    procedure :: init => lumi_linker_data_init
<SF lumi linker: procedures>≡
    subroutine lumi_linker_data_init &
        (data, model, flv, sqrts)
        class(lumi_linker_data_t), intent(out) :: data
        type(model_t), intent(in), target :: model
        type(flavor_t), dimension(2), intent(in) :: flv
        real(default), intent(in) :: sqrts
        data%model => model
        data%flv_in = flv
        data%pdg_in = flavor_get_pdg (data%flv_in)
        data%sqrts = sqrts
        select case (char (model_get_name (data%model)))
        case ("QCD","Test")
            call msg_fatal ("Lumi linker not available for model " &
                // char (model_get_name (data%model)) // ".")
        end select
    end subroutine lumi_linker_data_init

```

Output

```

<SF lumi linker: lumi linker data: TBP>+≡
    procedure :: write => lumi_linker_data_write
<SF lumi linker: procedures>+≡
    subroutine lumi_linker_data_write (data, unit, verbose)
        class(lumi_linker_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "Lumi linker data:"
        write (u, "(3x,A,A,A,A)") " prt_in = ", &
            char (flavor_get_name (data%flv_in(1))), &
            ", ", char (flavor_get_name (data%flv_in(2)))
        write (u, "(1x,A,ES19.12)") " sqrts = ", data%sqrts
    end subroutine lumi_linker_data_write

```

The number of parameters is two, collinear splitting for the two beams.

```

<SF lumi linker: lumi linker data: TBP>+≡
    procedure :: get_n_par => lumi_linker_data_get_n_par
<SF lumi linker: procedures>+≡
    function lumi_linker_data_get_n_par (data) result (n)
        class(lumi_linker_data_t), intent(in) :: data
        integer :: n
        n = 2
    end function lumi_linker_data_get_n_par

```

Return the outgoing particles PDG codes. This is either the incoming particle (if a photon is radiated), or the photon if that is the particle of the hard interaction. The latter is determined via the `photon` flag. There are two entries for the two beams (abuse?).

```

<SF lumi linker: lumi linker data: TBP>+≡
  procedure :: get_pdg_out => lumi_linker_data_get_pdg_out

<SF lumi linker: procedures>+≡
  !!! JRR: WK please check
  function lumi_linker_data_get_pdg_out (data) result (pdg_out)
    class(lumi_linker_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    integer :: i, n
    n = 2
    allocate (pdg_out (n))
    do i = 1, n
      pdg_out(i) = data%pdg_in(i)
    end do
  end function lumi_linker_data_get_pdg_out

```

Allocate the interaction record.

```

<SF lumi linker: lumi linker data: TBP>+≡
  procedure :: allocate_sf_int => lumi_linker_data_allocate_sf_int

<SF lumi linker: procedures>+≡
  subroutine lumi_linker_data_allocate_sf_int (data, sf_int)
    class(lumi_linker_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (lumi_linker_t :: sf_int)
  end subroutine lumi_linker_data_allocate_sf_int

```

### 10.8.3 The Lumi Linker object

This is a  $2 \rightarrow 4$  interaction, where, depending on the parameters, any two of the four outgoing particles are connected to the hard interactions, the others are radiated. Knowing that all particles are colorless, we do not have to deal with color.

The flavors are sorted such that the first two particles are the incoming leptons, the next two are the radiated particles, and the last two are the partons initiating the hard interaction. !!! JRR: WK please check I don't know actually whether this really fits into the setup done by WK.

```

<SF lumi linker: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: lumi_linker_t
    type(lumi_linker_data_t), pointer :: data => null ()
    real(default) :: x = 0
    real(default) :: q = 0
  contains
    <SF lumi linker: lumi linker: TBP>
  end type lumi_linker_t

```

Type string: At the moment a dummy routine.

```

<SF lumi linker: lumi linker: TBP>≡
  procedure :: type_string => lumi_linker_type_string

<SF lumi linker: procedures>+≡
  function lumi_linker_type_string (object) result (string)
    class(lumi_linker_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "Lumi linker: beamstrahlungs spectrum"
    else
      string = "Lumi linker: [undefined]"
    end if
  end function lumi_linker_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF lumi linker: lumi linker: TBP>+≡
  procedure :: write => lumi_linker_write

<SF lumi linker: procedures>+≡
  subroutine lumi_linker_write (object, unit)
    !!! JRR: WK please check
    !!! Guess these variables do not exist for the lumi linker (?)
    class(lumi_linker_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)") "SF parameters:"
        write (u, "(3x,A,ES17.10)") "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
          write (u, "(3x,A,ES17.10)") "Q =", object%q
        end if
      end if
      call object%base_write (u)
    else
      write (u, "(1x,A)") "Lumi linker data: [undefined]"
    end if
  end subroutine lumi_linker_write

```

#### 10.8.4 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF lumi linker: lumi linker: TBP>+≡
  procedure :: complete_kinematics => lumi_linker_complete_kinematics

```

*<SF lumi linker: procedures>+≡*

```
subroutine lumi_linker_complete_kinematics (sf_int, x, f, r, map)
  !!! JRR: WK please check
  !!! This cannot be correct, as the lumi linker structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(lumi_linker_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
    call msg_fatal ("Lumi linker: map flag not supported")
  else
    x(1) = r(1)
    f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
    sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
    sf_int%x = 0
    f = 0
  end select
end subroutine lumi_linker_complete_kinematics
```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

*<SF lumi linker: lumi linker: TBP>+≡*

```
procedure :: inverse_kinematics => lumi_linker_inverse_kinematics
```

*<SF lumi linker: procedures>+≡*

```
subroutine lumi_linker_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
  !!! JRR: WK please check
  !!! This cannot be correct, as the lumi linker structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(lumi_linker_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1
  logical :: set_mom
  set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
  if (map) then
    call msg_fatal ("Lumi linker: map flag not supported")
  else
    r(1) = x(1)
    f = 1
  end if
```



```

xb1 = 1 - x(1)
if (set_mom) then
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
    sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
    sf_int%x = 0
    f = 0
  end select
end if
end subroutine lumi_linker_inverse_kinematics

```

$\langle SF \text{ lumi linker: lumi linker: TBP} \rangle + \equiv$

```

procedure :: init => lumi_linker_init

```

$\langle SF \text{ lumi linker: procedures} \rangle + \equiv$

```

subroutine lumi_linker_init (sf_int, data)
  class(lumi_linker_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  logical, dimension(6) :: mask_h
  type(quantum_numbers_mask_t), dimension(6) :: mask
  integer, dimension(6) :: hel_lock
  type(polarization_t) :: pol1, pol2
  type(quantum_numbers_t), dimension(1) :: qn_fc1, qn_hel1, qn_fc2, qn_hel2
  type(flavor_t) :: flv_photon
  type(quantum_numbers_t) :: qn_photon, qn1, qn2
  type(quantum_numbers_t), dimension(6) :: qn
  type(state_iterator_t) :: it_hel1, it_hel2
  hel_lock = 0
  mask_h = .false.
  select type (data)
  type is (lumi_linker_data_t)
    hel_lock(1) = 5; hel_lock(5) = 1; mask_h(3) = .true.
    hel_lock(2) = 6; hel_lock(6) = 2; mask_h(4) = .true.
    mask = new_quantum_numbers_mask (.false., .false., mask_h)
    call sf_int%base_init (mask, [0._default, 0._default], &
      [0._default, 0._default], [0._default, 0._default], &
      hel_lock = hel_lock)
    sf_int%data => data
    call flavor_init (flv_photon, PHOTON, data%model)
    call quantum_numbers_init (qn_photon, flv_photon)
    call polarization_init_generic (pol1, data%flv_in(1))
    call quantum_numbers_init (qn_fc1(1), flv = data%flv_in(1))
    call polarization_init_generic (pol2, data%flv_in(2))
    call quantum_numbers_init (qn_fc2(1), flv = data%flv_in(2))
    call state_iterator_init (it_hel1, pol1%state)
    do while (state_iterator_is_valid (it_hel1))
      qn_hel1 = state_iterator_get_quantum_numbers (it_hel1)
      qn1 = qn_hel1(1) .merge. qn_fc1(1)
      qn(1) = qn1
      qn(3) = qn_photon; qn(5) = qn1
      call state_iterator_init (it_hel2, pol2%state)
      do while (state_iterator_is_valid (it_hel2))

```

```

        qn_hel2 = state_iterator_get_quantum_numbers (it_hel2)
        qn2 = qn_hel2(1) .merge. qn_fc2(1)
        qn(2) = qn2
        qn(4) = qn_photon; qn(6) = qn2
        call interaction_add_state (sf_int%interaction_t, qn)
        call state_iterator_advance (it_hel2)
    end do
    call state_iterator_advance (it_hel1)
end do
call polarization_final (pol1)
call polarization_final (pol2)
call interaction_freeze (sf_int%interaction_t)
!!! JRR: WK please check
!!! Is this correct if the photon is the radiating particle??
call sf_int%set_incoming ([1,2])
call sf_int%set_radiated ([3,4])
call sf_int%set_outgoing ([5,6])
sf_int%status = SF_INITIAL
end select
end subroutine lumi_linker_init

```

### 10.8.5 Lumi linker application

*(SF lumi linker: lumi linker: TBP)+≡*

```
procedure :: apply => lumi_linker_apply
```

*(SF lumi linker: procedures)+≡*

```

subroutine lumi_linker_apply (sf_int, scale) !!!, no_map)
!!! subroutine lumi_linker_apply (sf_int, r, no_map)
!!! JRR: WK please check
!!! this somehow is different from the PDFs, so the types
!!! do not match. Guess we need two types of strfun,
!!! 1->2 and 2->4, nez-pas?
class(lumi_linker_t), intent(inout) :: sf_int
!!! Dummy
real(default), intent(in) :: scale
!!!real(default), dimension(2), intent(in) :: r
!!!logical, intent(in) :: no_map
!!!type(vector4_t), dimension(2) :: k
!!!type(splitting_data_t), dimension(2) :: sd
!!!real(default), dimension(2) :: m_in, x, xb
real(default) :: f
!!!type(vector4_t), dimension(4) :: q
associate (data => sf_int%data)
!!! !!! JRR: WKK please check
!!! !!! No clue what to do here :(((
!!! k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
!!! k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
!!! m_in = data% m_in
!!! sd = new_splitting_data (k, m_in**2, m_in**2, m_in)
!!! call strfun (f, x, xb, r, lumi_linker_data, no_map)
!!! call splitting_set_t_bounds (sd, x, xb)
!!! call splitting_set_collinear (sd)

```

```

    !!! q((/1, 3/)) = split_momentum (k(1), sd(1))
    !!! q((/2, 4/)) = split_momentum (k(2), sd(2))
    !!! call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
end associate
call interaction_set_matrix_element &
    (sf_int%interaction_t, cmplx (f, kind=default))
sf_int%status = SF_EVALUATED
end subroutine lumi_linker_apply

```

## 10.9 Photon collider: CIRCE2

```
<sf_circe2.f90>≡  
  <File header>  
  
  module sf_circe2  
  
    <Use kinds>  
    <Use strings>  
    <Use file utils>  
    use diagnostics !NODEP!  
    use tao_random_numbers !NODEP!  
    use lorentz !NODEP!  
    use models  
    use flavors  
    use helicities  
    use quantum_numbers  
    use state_matrices  
    use polarizations  
    use interactions  
    use sf_aux  
    use sf_base  
    use circe2 !NODEP!  
  
    <Standard module head>  
  
    <SF circe2: public>  
  
    <SF circe2: types>  
  
    <SF circe2: variables>  
  
    contains  
  
    <SF circe2: procedures>  
  
  end module sf_circe2
```

### 10.9.1 Physics

CIRCE2 describes photon spectra Beamstrahlung is applied before ISR. The CIRCE2 implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously).

### 10.9.2 The CIRCE2 data block

The CIRCE2 parameters are: file and collider specification, incoming (= outgoing) particles. The luminosity is returned by `cir2lm`.

```
<SF circe2: public>≡  
  public :: circe2_data_t  
  
<SF circe2: types>≡  
  type, extends (sf_data_t) :: circe2_data_t
```

```

private
type(flavor_t), dimension(2) :: flv_in
integer, dimension(2) :: pdg = 0
real(default), dimension(2) :: mass = 0
logical :: generate = .true.
type(tao_random_state), pointer :: rng => null ()
logical :: map = .true.
integer, dimension(2) :: map_mode = -1
real(default), dimension(2) :: map_power = 0
type(string_t) :: file
type(string_t) :: design
real(default) :: sqrts = 0
logical :: polarized = .false.
real(default) :: lumi = 0
real(default), dimension(-1:1,-1:1) :: lumi_hel_frac = 0
real(default), dimension(0:4) :: lumi_hel_sum = 0
contains
  <SF circe2: circe2 data: TBP>
end type circe2_data_t

<SF circe2: circe2 data: TBP>≡
  procedure :: init => circe2_data_init

<SF circe2: procedures>≡
  subroutine circe2_data_init &
    (data, flv_in, generate, rng, map, file, design, sqrts, polarized)
    class(circe2_data_t), intent(out) :: data
    type(flavor_t), dimension(2), intent(in) :: flv_in
    logical, intent(in) :: generate
    type(tao_random_state), intent(in), target :: rng
    logical, intent(in) :: map
    type(string_t), intent(in) :: file, design
    real(default), intent(in) :: sqrts
    logical, intent(in) :: polarized
    integer :: error, i, h1, h2, h
    data%flv_in = flv_in
    data%pdg = flavor_get_pdg (data%flv_in)
    data%mass = flavor_get_mass (data%flv_in)
    data%generate = generate
    data%rng => rng
    data%map = map
    if (data%map) then
      do i = 1, 2
        select case (abs (data%pdg(i)))
          case (PHOTON); data%map_mode(i) = 0; data%map_power(i) = 3
          case (ELECTRON); data%map_mode(i) = 1; data%map_power(i) = 12
          case default; call msg_fatal &
            ("CIRCE2: defined only for photon and electron beams")
        end select
      end do
    else
      data%map_mode = -1
    end if
    data%file = file

```

```

data%design = design
data%sqrts = sqrts
data%polarized = polarized
error = 1
call cir2ld (trim (char(data%file)), trim (char(data%design)), &
            dble (data%sqrts), error)
select case (error)
case (-1)
    call msg_fatal ("CIRCE2: data file not found.")
case (-2)
    call msg_fatal ("CIRCE2: beam parameters do not match data file.")
case (-3)
    call msg_fatal ("CIRCE2: invalid format of data file.")
case (-4)
    call msg_fatal ("CIRCE2: data file too large.")
end select
data%lumi = cir2lm (data%pdg(1), 0, data%pdg(2), 0)
if (data%lumi == 0) then
    call circe2_data_write (data)
    call msg_fatal ("CIRCE2: luminosity vanishes for specified beams.")
end if
if (data%polarized) then
    h = 0
    do h1 = -1, 1, 2
        do h2 = -1, 1, 2
            data%lumi_hel_frac(h1,h2) = &
                cir2lm (data%pdg(1), h1, data%pdg(2), h2) / data%lumi
            h = h + 1
            data%lumi_hel_sum(h) = &
                data%lumi_hel_sum(h-1) + data%lumi_hel_frac(h1,h2)
        end do
    end do
    data%lumi_hel_sum(4) = 1
end if
end subroutine circe2_data_init

```

Output

```

<SF circe2: circe2 data: TBP>+≡
    procedure :: write => circe2_data_write

<SF circe2: procedures>+≡
    subroutine circe2_data_write (data, unit, verbose) !, md5)
        class(circe2_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: h1, h2
        integer :: u
        !!! JRR: WK please check: still needed?
    !    logical, intent(in), optional :: md5
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "CIRCE2 data:"
        write (u, "(3x,A,A)")      "  file = ", char(data%file)
        write (u, "(3x,A,A)")      "  design = ", char(data%design)
        write (u, "(1x,A,ES19.12)") "  sqrts = ", data%sqrts
    end subroutine

```

```

write (u, "(3x,A,A,A,A)") " prt_in = ", &
char (flavor_get_name (data%flv_in(1))), &
", ", char (flavor_get_name (data%flv_in(2)))
write (u, "(1x,A,ES19.12)") " mass = ", data%mass
write (u, "(3x,A,L1)") " polarized = ", data%polarized
write (u, "(1x,A,ES19.12)") " luminosity = ", data%lumi
if (data%polarized) then
do h1 = -1, 1, 2
do h2 = -1, 1, 2
write (u, "(6x,'(,I2,1x,I2,')',1x,'=',1x)", advance="no") h1, h2
write (u, "(6x, ES19.12)") data%lumi_hel_frac(h1,h2)
end do
end do
end if
write (u, "(3x,A,L1)") " generate = ", data%generate
write (u, "(3x,A,L1)") " map = ", data%map
if (data%map) then
write (u, "(3x,A,I3)") " mode = ", data%map_mode
write (u, "(3x,A,I3)") " power = ", data%map_power
end if
end subroutine circe2_data_write

```

The number of parameters is two, collinear splitting for the two beams.

```

<SF circe2: circe2 data: TBP>+=
procedure :: get_n_par => circe2_data_get_n_par

<SF circe2: procedures>+=
function circe2_data_get_n_par (data) result (n)
class(circe2_data_t), intent(in) :: data
integer :: n
n = 2
end function circe2_data_get_n_par

```

Return the outgoing particles PDG codes. For CIRCE2, no state matrix is introduced, it is a  $2 \rightarrow 2$  scattering with the outgoing particles identical to the incoming ones (cf. above). There are two components for the two beams.

```

<SF circe2: circe2 data: TBP>+=
procedure :: get_pdg_out => circe2_data_get_pdg_out

<SF circe2: procedures>+=
!!! JRR: WK please check
function circe2_data_get_pdg_out (data) result (pdg_out)
class(circe2_data_t), intent(in) :: data
integer, dimension(:), allocatable :: pdg_out
integer :: i, n
n = 2
allocate (pdg_out (n))
do i = 1, n
pdg_out(i) = data%pdg(i)
end do
end function circe2_data_get_pdg_out

```

Allocate the interaction record.

```

<SF circe2: circe2 data: TBP>+=
procedure :: allocate_sf_int => circe2_data_allocate_sf_int

```

```

<SF circe2: procedures>+≡
  subroutine circe2_data_allocate_sf_int (data, sf_int)
    class(circe2_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (circe2_t :: sf_int)
  end subroutine circe2_data_allocate_sf_int

```

### 10.9.3 The CIRCE2 object

For CIRCE2 spectra it does not make sense to describe the state matrix as a radiation interaction, even if photons originate from laser backscattering. Instead, it is a  $2 \rightarrow 2$  interaction where the incoming particles are identical to the outgoing ones.

The current implementation of CIRCE2 does support polarization and classical correlations, but no entanglement, so the density matrix of the outgoing particles is diagonal. The incoming particles are unpolarized (user-defined polarization for beams is meaningless, since polarization is described by the data file). The outgoing particles are polarized or polarization-averaged, depending on user request.

When assigning matrix elements, we scan the previously initialized state matrix. For each entry, we extract helicity and call the structure function. In the unpolarized case, the helicity is undefined and replaced by value zero. In the polarized case, there are four entries. If the generator is used, only one entry is nonzero in each call. Which one, is determined by comparing with a previously (randomly, distributed by relative luminosity) selected pair of helicities.

```

<SF circe2: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: circe2_t
    type(circe2_data_t), pointer :: data => null ()
    real(default) :: x = 0
    real(default) :: q = 0
  contains
    <SF circe2: circe2: TBP>
  end type circe2_t

```

Type string: show file and design of CIRCE2 structure function.

```

<SF circe2: circe2: TBP>≡
  procedure :: type_string => circe2_type_string

<SF circe2: procedures>+≡
  function circe2_type_string (object) result (string)
    !!! JRR: WK please check
    class(circe2_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "CIRCE2: " // object%data%file // ", " // object%data%design
    else
      string = "CIRCE2: [undefined]"
    end if
  end function circe2_type_string

```



Output. Call the interaction routine after displaying the configuration.

```

<SF circe2: circe2: TBP>+≡
  procedure :: write => circe2_write

<SF circe2: procedures>+≡
  subroutine circe2_write (object, unit)
    !!! JRR: WK please check
    !!! Guess these variables do not exist for CIRCE2 (?)
    class(circe2_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)") "SF parameters:"
        write (u, "(3x,A,ES17.10)") "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
          write (u, "(3x,A,ES17.10)") "Q =", object%q
        end if
      end if
    end if
    call object%base_write (u)
  else
    write (u, "(1x,A)") "CIRCE2 data: [undefined]"
  end if
end subroutine circe2_write

```

#### 10.9.4 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF circe2: circe2: TBP>+≡
  procedure :: complete_kinematics => circe2_complete_kinematics

<SF circe2: procedures>+≡
  subroutine circe2_complete_kinematics (sf_int, x, f, r, map)
    !!! JRR: WK please check
    !!! This cannot be correct, as the lumi linker structure function has
    !!! twice the variables (2->4 instead of 1->2 splitting)
    class(circe2_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    logical, intent(in) :: map
    real(default) :: xb1
    if (map) then
      call msg_fatal ("CIRCE2: map flag not supported")
    else
      x(1) = r(1)
      f = 1
    end if
  end subroutine

```

```

xb1 = 1 - x(1)
call sf_int%split_momentum (x, xb1)
select case (sf_int%status)
case (SF_DONE_KINEMATICS)
  sf_int%x = x(1)
case (SF_FAILED_KINEMATICS)
  sf_int%x = 0
  f = 0
end select
end subroutine circe2_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF circe2: circe2: TBP>+≡
  procedure :: inverse_kinematics => circe2_inverse_kinematics

<SF circe2: procedures>+≡
  subroutine circe2_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
    !!! JRR: WK please check
    !!! This cannot be correct, as the CIRCE2 structure function has
    !!! twice the variables (2->4 instead of 1->2 splitting)
    class(circe2_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default) :: xb1
    logical :: set_mom
    set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
    if (map) then
      call msg_fatal ("CIRCE2: map flag not supported")
    else
      r(1) = x(1)
      f = 1
    end if
    xb1 = 1 - x(1)
    if (set_mom) then
      call sf_int%split_momentum (x, xb1)
      select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
      case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
      end select
    end if
  end subroutine circe2_inverse_kinematics

  <SF circe2: circe2: TBP>+≡
    procedure :: init => circe2_init

```

*<SF circe2: procedures>+≡*

```

subroutine circe2_init (sf_int, data)
  !!! JRR: WK please check
  class(circe2_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(polarization_t) :: pol3, pol4
  real(default), dimension(2), parameter :: pol_init = 0.5_default
  logical, dimension(4) :: mask_h
  real(default), dimension(0) :: null_array
  type(quantum_numbers_mask_t), dimension(4) :: mask
  type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
  type(state_iterator_t) :: it_hel3, it_hel4
  select type (data)
  type is (circe2_data_t)
    mask_h(1:2) = .true.
    mask_h(3:4) = .not. data%polarized
    mask = new_quantum_numbers_mask (.false., .false., mask_h)
    call quantum_numbers_init (qn_fc(1), flv = data%flv_in(1))
    call quantum_numbers_init (qn_fc(2), flv = data%flv_in(2))
    call quantum_numbers_init (qn_fc(3), flv = data%flv_in(1))
    call quantum_numbers_init (qn_fc(4), flv = data%flv_in(2))
    if (data%polarized) then
      call polarization_init_diagonal &
        (pol3, data%flv_in(1), pol_init)
      call polarization_init_diagonal &
        (pol4, data%flv_in(1), pol_init)
    else
      call polarization_init_unpolarized (pol3, data%flv_in(1))
      call polarization_init_unpolarized (pol4, data%flv_in(1))
    end if
    call sf_int%base_init (mask, [0._default, 0._default], &
      null_array, [0._default, 0._default])
    sf_int%data => data
    call state_iterator_init (it_hel3, pol3%state)
    qn(1) = qn_fc(1)
    qn(2) = qn_fc(2)
    do while (state_iterator_is_valid (it_hel3))
      qn_hel(3:3) = state_iterator_get_quantum_numbers (it_hel3)
      qn(3) = qn_hel(3) .merge. qn_fc(3)
      call state_iterator_init (it_hel4, pol4%state)
      do while (state_iterator_is_valid (it_hel4))
        qn_hel(4:4) = state_iterator_get_quantum_numbers (it_hel4)
        qn(4) = qn_hel(4) .merge. qn_fc(4)
        call interaction_add_state (sf_int%interaction_t, qn)
        call state_iterator_advance (it_hel4)
      end do
      call state_iterator_advance (it_hel3)
    end do
    call polarization_final (pol3)
    call polarization_final (pol4)
    call interaction_freeze (sf_int%interaction_t)
    !!! JRR: WK please check
    call sf_int%set_incoming ([1,2])
    call sf_int%set_outgoing ([3,4])
  end select
end subroutine

```

```

        sf_int%status = SF_INITIAL
    end select
end subroutine circe2_init

```

### 10.9.5 CIRCE2 structure function

Compute the CIRCE2 structure function: either using the generator or the structure function value. In the latter case, we may employ a power mapping for the singular endpoint that uses some heuristics for the mapping parameters.

The generator mode is the default and by far more useful, since there is virtually no loss in accuracy or reweighting efficiency for a process with CIRCE2 and without. The formal problem with this approach is the indeterministic integration, which is presumably harmless since the corresponding integration dimensions factorize in the VEGAS approach.

In generator mode with polarized beams, for each event only one helicity is selected. (See `interaction_apply_circe2` below.) The selection is based on the relative luminosities. Hence, the function value (unity) is multiplied by the number of helicity states (four).

Similarly, in non-generator mode, the luminosity densities for the helicity states are individually normalized and must be multiplied by their luminosity fraction and multiplied by four.

If `no_map` is set, switch off generator and mapping modes.

```

<CIRCE2: procedures>≡
subroutine strfun (f, x, r, hel, data, no_map)
    real(default), intent(out) :: f
    real(default), dimension(2), intent(out) :: x
    real(default), dimension(2), intent(in) :: r
    real(default), dimension(2) :: fi
    integer, dimension(2), intent(in) :: hel
    double precision, dimension(2) :: xx
    type(circe2_data_t), intent(in) :: data
    logical, intent(in) :: no_map
    if (data%generate .and. .not. no_map) then
        rng_tmp => data%rng
        call cir2gn &
            (data%pdg(1), hel(1), data%pdg(2), hel(2), xx(1), xx(2), rn_sub)
        x = xx
        if (data%polarized) then
            f = 4
        else
            f = 1
        end if
    else
        if (no_map) then
            x = r
            fi = 1
        else
            call circe2_map (fi, x, r, data%map_mode, data%map_power)
        end if
        f = product (fi) &
            * cir2dn (data%pdg(1), hel(1), data%pdg(2), hel(2), &

```

```

                                dble (x(1)), dble (x(2)))
    if (data%polarized) then
        f = 4 * f * data%lumi_hel_frac (hel(1), hel(2))
    end if
end if
end subroutine strfun

```

The CIRCE2 calls need the generator as argument, but not the generator state. Thus, we temporarily assign a pointer to the state to a module variable which is accessed by the generator.

*(CIRCE2: variables)*≡  
 type(tao\_random\_state), pointer :: rng\_tmp => null ()

Here is the generator call in the form that CIRCE2 expects:

*(CIRCE2: procedures)*+≡  
 subroutine rn\_sub (r)  
   double precision, intent(out) :: r  
   real(default) :: x  
   call tao\_random\_number (rng\_tmp, x)  
   r = x  
end subroutine rn\_sub

Map the random numbers to  $x$  values, using specified power mappings for both beams. This function applies to a single beam but is elemental, so it is applied to the beam pair simultaneously.

We define power mappings (modes) for  $x = 0$  or, alternatively, for  $x = 1$ . Mode  $x = -1$  means no mapping.

Near zero (photon case) a power 2...3 should be sufficient to cancel the singularity, while near one (electron) the very narrow smeared delta function requires a high power  $\sim 10$  to be identified. Quite generically, if a low  $x$  value  $x_0$  needs to be resolved by the mapping, a power of the order  $-\frac{1}{2} \log x_0$  is appropriate.

*(CIRCE2: procedures)*+≡  
 elemental subroutine circe2\_map (f, x, r, mode, power)  
   real(default), intent(out) :: f, x  
   real(default), intent(in) :: r  
   integer, intent(in) :: mode  
   real(default), intent(in) :: power  
   real(default) :: xbar, rbar  
   select case (mode)  
   case (0)  
     if (r <= tiny(1.\_default)) then  
       x = 0  
       f = 0  
     else  
       x = r \*\* power  
       f = power \* x / r  
     end if  
   case (1)  
     rbar = 1 - r  
     if (rbar <= tiny(1.\_default)) then  
       xbar = 0

```

        f = 0
    else
        xbar = rbar ** power
        f = power * xbar / rbar
    end if
    x = 1 - xbar
case default
    x = r
    f = 1
end select
end subroutine circe2_map

```

### 10.9.6 CIRCE2 application

This function works on both beams, assuming that the input momentum has been set. We apply the usual splitting formalism, specializing to collinear splitting, but throw away the radiated momenta.

```

<SF circe2: circe2: TBP>+≡
    procedure :: apply => circe2_apply

<SF circe2: procedures>+≡
    subroutine circe2_apply (sf_int, scale) !!!r, circe2_data, no_map)
        !!! JRR: WK please check
        !!! this somehow is different from the PDFs, so how
        !!! do we connect things?
        class(circe2_t), intent(inout) :: sf_int
        real(default), intent(in) :: scale
        !!! real(default), dimension(2), intent(in) :: r
        !!! type(circe2_data_t), intent(in) :: circe2_data
        !!! logical, intent(in) :: no_map
        !!! complex(default), parameter :: CZERO = 0
        !!! type(vector4_t), dimension(2) :: k
        !!! integer, dimension(2) :: h, h_gen, h_tmp
        !!! type(state_iterator_t) :: it
        !!! real(default) :: f
        !!! real(default), dimension(2) :: x
        !!! type(splitting_data_t), dimension(2) :: sd
        !!! type(vector4_t), dimension(4) :: q
        !!! k(1) = interaction_get_momentum (int, 1)
        !!! k(2) = interaction_get_momentum (int, 2)
        associate (data => sf_int%data)
        !!! if (circe2_data%generate) then
        !!!     call interaction_set_matrix_element (int, CZERO)
        !!!     if (circe2_data%polarized) then
        !!!         call circe2_select_hel &
        !!!             (h_gen, circe2_data%rng, circe2_data%lumi_hel_sum)
        !!!     else
        !!!         h_gen = 0
        !!!     end if
        !!! end if
        !!! call state_iterator_init (it, interaction_get_state_matrix_ptr (int))
        !!! LOOP_HEL: do while (state_iterator_is_valid (it))
        !!!     if (circe2_data%polarized) then

```

```

!!!      h_tmp = helicity_get (state_iterator_get_helicity (it, 3))
!!!      h(1) = h_tmp(1)
!!!      h_tmp = helicity_get (state_iterator_get_helicity (it, 4))
!!!      h(2) = h_tmp(1)
!!!      else
!!!          h = 0
!!!      end if
!!!      if (.not. circe2_data%generate .or. all (h == h_gen)) then
!!!          call strfun (f, x, r, h, circe2_data, no_map)
!!!          call state_iterator_set_matrix_element (it, cmplx (f, kind=default))
!!!      end if
!!!      call state_iterator_advance (it)
!!! end do LOOP_HEL
!!! sd = new_splitting_data &
!!!      (k, circe2_data%mass**2, 0._default, circe2_data%mass)
!!! call splitting_set_t_bounds (sd, x, 1-x)
!!! call splitting_set_collinear (sd)
!!! q((/1, 3/)) = split_momentum (k(1), sd(1))
!!! q((/2, 4/)) = split_momentum (k(2), sd(2))
!!! call interaction_set_momenta (int, q(3:4), outgoing=.true.)
end associate
sf_int%status = SF_EVALUATED
end subroutine circe2_apply

```

Select a channel for the generator mode. We could use `cir2ch`, but this selects flavor and helicity; we just want to select helicity.

*(CIRCE2: procedures)+≡*

```

subroutine circe2_select_hel (hel, rng, threshold)
    integer, dimension(2), intent(out) :: hel
    type(tao_random_state), pointer :: rng
    real(default), dimension(0:4) :: threshold
    real(default) :: x
    integer :: h1, h2
    integer :: h
    call tao_random_number (rng, x)
    h = 0
    do h1 = -1, 1, 2
        do h2 = -1, 1, 2
            h = h + 1
            if (x <= threshold(h)) then
                hel(1) = h1; hel(2) = h2; return
            end if
        end do
    end do
    call msg_bug ("CIRCE2: helicity selection failed")
end subroutine circe2_select_hel

```

## 10.10 Energy-scan spectrum

This spectrum is actually a trick that allows us to plot the c.m. energy dependence of a cross section without scanning the input energy. We start with the observation that a spectrum  $f(x)$ , applied to one of the incoming beams only, results in a cross section

$$\sigma = \int dx f(x) \hat{\sigma}(xs). \quad (10.75)$$

We want to compute the distribution of  $E = \sqrt{\hat{s}} = \sqrt{xs}$ , i.e.,

$$\frac{d\sigma}{dE} = \frac{2\sqrt{x}}{\sqrt{s}} \frac{d\sigma}{dx} = \frac{2\sqrt{x}}{\sqrt{s}} f(x) \hat{\sigma}(xs), \quad (10.76)$$

so if we set

$$f(x) = \frac{\sqrt{s}}{2\sqrt{x}}, \quad (10.77)$$

we get the distribution

$$\frac{d\sigma}{dE} = \hat{\sigma}(\hat{s} = E^2). \quad (10.78)$$

If we want to start, in the first iteration, from a homogeneous energy scan, we should apply a mapping

$$x = y^2 \quad \text{with} \quad dx = 2\sqrt{x} dy, \quad (10.79)$$

which cancels out  $f(x)$  apart from the numerator  $\sqrt{s}$ .

The input parameter for this spectrum applies to the energy directly. Note that the spectrum must not be applied to more than one beam, and that the total cross section is essentially meaningless (it is the area under the energy distribution).

A direct application of this procedure would, of course, result in boosted events since only one beam energy is scaled down. However, we can achieve the same effect without a boost if we scale both  $x$  values such that their product remains constant, i.e., instead of generating  $x$  and applying it for one beam, we apply  $\sqrt{x}$  for both beams.

```
<sf_escan.f90>≡
  <File header>
```

```
module sf_escan
```

```
  <Use kinds>
```

```
  <Use strings>
```

```
  <Use file utils>
```

```
    use diagnostics !NODEP!
```

```
    use lorentz !NODEP!
```

```
    use flavors
```

```
    use helicities
```

```
    use colors
```

```
    use quantum_numbers
```

```
    use state_matrices
```

```
    use polarizations
```

```
    use interactions
```



```

    use sf_aux
    use sf_base

    <Standard module head>

    <SF escan: public>

    <SF escan: types>

contains

    <SF escan: procedures>

end module sf_escan

```

### 10.10.1 Data type

```

<SF escan: public>≡
    public :: escan_data_t

<SF escan: types>≡
    type, extends(sf_data_t) :: escan_data_t
        private
            logical, dimension(2) :: affects_beam
            type(flavor_t), dimension(2) :: flv
            real(default), dimension(2) :: mass = 0
            real(default) :: sqrts = 0
        contains
            <SF escan: escan data: TBP>
        end type escan_data_t

<SF escan: escan data: TBP>≡
    procedure :: init => escan_data_init

<SF escan: procedures>≡
    subroutine escan_data_init (data, affects_beam, flv_in, sqrts)
        class(escan_data_t), intent(out) :: data
        logical, dimension(2), intent(in) :: affects_beam
        type(flavor_t), dimension(2), intent(in) :: flv_in
        real(default), intent(in) :: sqrts
        data%affects_beam = affects_beam
        data%flv = flv_in
        data%mass = flavor_get_mass (data%flv)
        data%sqrts = sqrts
    end subroutine escan_data_init

```

Output

```

<SF escan: escan data: TBP>+≡
    procedure :: write => escan_data_write

<SF escan: procedures>+≡
    subroutine escan_data_write (data, unit, verbose) !, md5)
        class(escan_data_t), intent(in) :: data
        integer, intent(in), optional :: unit

```

```

        logical, intent(in), optional :: verbose
        integer :: u
        !!! JRR: WK please check: do we need this here?
!       logical, intent(in), optional :: md5
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "Energy-scan data:"
        write (u, "(3x,A,A,A,A)") &
            " prt_in = ", char (flavor_get_name (data%flv(1))), &
            ", ", char (flavor_get_name (data%flv(2)))
    end subroutine escan_data_write

```

JRR: WK please check. The number of parameters is equal to the number of entries in the `affects_beam` flag. Kinematics is completely collinear.

```

<SF escan: escan data: TBP>+≡
    procedure :: get_n_par => escan_data_get_n_par
<SF escan: procedures>+≡
    function escan_data_get_n_par (data) result (n)
        !!! JRR: WK please check
        !!! What about correlation of x values?
        class(escan_data_t), intent(in) :: data
        integer :: n
        n = count (data%affects_beam)
    end function escan_data_get_n_par

```

Return the outgoing particles PDG codes. This is always the same as the incoming particle, where we use two indices for the two beams.

```

<SF escan: escan data: TBP>+≡
    procedure :: get_pdg_out => escan_data_get_pdg_out
<SF escan: procedures>+≡
    !!! JRR: WK please check
    function escan_data_get_pdg_out (data) result (pdg_out)
        class(escan_data_t), intent(in) :: data
        integer, dimension(:), allocatable :: pdg_out
        integer :: i, n
        n = 2
        allocate (pdg_out (n))
        do i = 1, n
            pdg_out(i) = flavor_get_pdg (data%flv(i))
        end do
    end function escan_data_get_pdg_out

```

Allocate the interaction record.

```

<SF escan: escan data: TBP>+≡
    procedure :: allocate_sf_int => escan_data_allocate_sf_int
<SF escan: procedures>+≡
    subroutine escan_data_allocate_sf_int (data, sf_int)
        class(escan_data_t), intent(in) :: data
        class(sf_int_t), intent(inout), allocatable :: sf_int
        allocate (escan_t :: sf_int)
    end subroutine escan_data_allocate_sf_int

```

## 10.10.2 The Energy-scan object

This is a spectrum, not a radiation. Depending on whether one or both beams are involved, we create an interaction with 1 (2) incoming and 1 (2) outgoing particles, flavor, color, and helicity being carried through. So,  $x$  and  $q$  are 2-dimensional and have to be restricted to one dimension if only one beam is affected.

```

<SF escan: types>+≡
  !!! JRR: WK please check
  type, extends (sf_int_t) :: escan_t
    type(escan_data_t), pointer :: data => null ()
    real(default), dimension(2) :: x = 0
    real(default), dimension(2) :: q = 0
  contains
    <SF escan: escan: TBP>
  end type escan_t

```

Type string: for the energy scan this is just a dummy function.

```

<SF escan: escan: TBP>≡
  procedure :: type_string => escan_type_string

<SF escan: procedures>+≡
  function escan_type_string (object) result (string)
    !!! JRR: WK please check
    class(escan_t), intent(in) :: object
    type(string_t) :: string
    if (associated (object%data)) then
      string = "Escan: energy scan"
    else
      string = "Escan: [undefined]"
    end if
  end function escan_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF escan: escan: TBP>+≡
  procedure :: write => escan_write

<SF escan: procedures>+≡
  subroutine escan_write (object, unit)
    !!! JRR: WK please check
    !!! Guess these variables do not exist for CIRCE1 (?)
    class(escan_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)") "SF parameters:"
        write (u, "(3x,A,ES17.10)") "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
          write (u, "(3x,A,ES17.10)") "Q =", object%q
        end if
      end if
    end if
  end subroutine

```

```

        call object%base_write (u)
    else
        write (u, "(1x,A)") "Energy scan data: [undefined]"
    end if
end subroutine escan_write

```

### 10.10.3 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF escan: escan: TBP>+=
    procedure :: complete_kinematics => escan_complete_kinematics

<SF escan: procedures>+=
    subroutine escan_complete_kinematics (sf_int, x, f, r, map)
        class(escan_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(out) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: r
        logical, intent(in) :: map
        real(default) :: xb1
        if (map) then
            call msg_fatal ("Energy scan: map flag not supported")
        else
            x(1) = r(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        call sf_int%split_momentum (x, xb1)
        select case (sf_int%status)
        case (SF_DONE_KINEMATICS)
            sf_int%x = x(1)
        case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
        end select
    end subroutine escan_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF escan: escan: TBP>+=
    procedure :: inverse_kinematics => escan_inverse_kinematics

<SF escan: procedures>+=
    subroutine escan_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        class(escan_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r

```

```

logical, intent(in) :: map
logical, intent(in), optional :: set_momenta
real(default) :: xb1
logical :: set_mom
set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
if (map) then
    call msg_fatal ("Energy scan: map flag not supported")
else
    r(1) = x(1)
    f = 1
end if
xb1 = 1 - x(1)
if (set_mom) then
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end if
end subroutine escan_inverse_kinematics

```

*<SF escan: escan: TBP>+≡*

```

procedure :: init => escan_init

```

*<SF escan: procedures>+≡*

```

subroutine escan_init (sf_int, data)
    class(escan_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(4) :: mask
    integer, dimension(4) :: hel_lock
    real(default), dimension(0) :: null_array
    type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
    type(polarization_t) :: pol1, pol2
    type(state_iterator_t) :: it_hel1, it_hel2
    integer :: i
    select type (data)
    type is (escan_data_t)
        if (all (data%affects_beam)) then
            hel_lock = (/ 3, 4, 1, 2 /)
            call sf_int%base_init (mask, data%mass**2, &
                null_array, data%mass**2, hel_lock = hel_lock)
        else if (data%affects_beam(1)) then
            hel_lock = (/ 2, 1, 0, 0 /)
            call sf_int%base_init (mask(1:2), [data%mass(1)**2], &
                null_array, [data%mass(1)**2], hel_lock = hel_lock(1:2))
        else if (data%affects_beam(2)) then
            hel_lock = (/ 2, 1, 0, 0 /)
            !!! JRR: WK please check
            !!! Seems to be a cut-and-paste error, is it really
            !!! identical to the case affects_beam(1) ??
            call sf_int%base_init (mask(1:2), [data%mass(2)**2], &

```

```

        null_array, [data%mass(2)**2], hel_lock = hel_lock(1:2))
end if
sf_int%data => data
do i = 1, 2
    call quantum_numbers_init (qn_fc(i), &
        flv = data%flv(i), col = color_from_flavor (data%flv(i)))
    call quantum_numbers_init (qn_fc(i+2), &
        flv = data%flv(i), col = color_from_flavor (data%flv(i)))
end do
call polarization_init_generic (pol1, data%flv(1))
call state_iterator_init (it_hel1, pol1%state)
do while (state_iterator_is_valid (it_hel1))
    qn_hel(1:1) = state_iterator_get_quantum_numbers (it_hel1)
    qn_hel(3:3) = state_iterator_get_quantum_numbers (it_hel1)
    call polarization_init_generic (pol2, data%flv(2))
    call state_iterator_init (it_hel2, pol2%state)
    do while (state_iterator_is_valid (it_hel2))
        qn_hel(2:2) = state_iterator_get_quantum_numbers (it_hel2)
        qn_hel(4:4) = state_iterator_get_quantum_numbers (it_hel2)
        qn = qn_hel .merge. qn_fc
        if (all (data%affects_beam)) then
            call interaction_add_state (sf_int%interaction_t, qn)
        else if (data%affects_beam(1)) then
            call interaction_add_state (sf_int%interaction_t, qn((/1,3/)))
        else if (data%affects_beam(2)) then
            call interaction_add_state (sf_int%interaction_t, qn((/2,4/)))
        end if
        call state_iterator_advance (it_hel2)
    end do
    call polarization_final (pol2)
    call state_iterator_advance (it_hel1)
end do
call polarization_final (pol2)
call interaction_freeze (sf_int%interaction_t)
!!! JRR: WK please check
call sf_int%set_incoming ([1,2])
call sf_int%set_outgoing ([3,4])
sf_int%status = SF_INITIAL
end select
end subroutine escan_init

```

#### 10.10.4 Energy-scan structure function

This is the structure function for a single beam, suitably mapped.

As stated above, the mapping cancels the denominator of the structure function, so the factor is a constant. The normalization is determined such that the integral (the 'cross section' returned by WHIZARD) is unity for a unit test matrix element.

*(Escan: procedures)*≡

```

subroutine strfun (f, x, r, data)
    real(default), intent(out) :: f, x
    real(default), intent(in) :: r

```

```

    type(escan_data_t), intent(in) :: data
    x = r**2
    f = 1
end subroutine strfun

```

### 10.10.5 Energy scan application

Depending on the affected beams, we either apply the structure function to one beam, or to both beams using the square root of  $x_i$ .

There is no `no_map` flag involved since there is no mapping anyway.

```

<SF escan: escan: TBP>+=
  procedure :: apply => escan_apply

<SF escan: procedures>+=
  subroutine escan_apply (sf_int, scale) !!! r, data)
    !!! JRR: WK please check, different from the PDFs
    class(escan_t), intent(inout) :: sf_int
    !!! Dummy
    real(default), intent(in) :: scale
    !!! real(default), dimension(1), intent(in) :: r
    !!! type(escan_data_t), intent(in) :: data
    !!! type(vector4_t), dimension(2) :: k
    real(default) :: f, xx
    !!! real(default), dimension(2) :: x
    !!! type(splitting_data_t), dimension(2) :: sd
    !!! type(vector4_t), dimension(4) :: q
    associate (data => sf_int%data)
      !!! k(1) = interaction_get_momentum (int, 1)
      !!! if (all (data%affects_beam)) then
      !!!   k(2) = interaction_get_momentum (int, 2)
      !!! else
      !!!   k(2) = k(1)
      !!! end if
      !!! call strfun (f, xx, r(1), data)
      !!! if (all (data%affects_beam)) then
      !!!   x = sqrt (xx)
      !!! else
      !!!   where (data%affects_beam)
      !!!     x = xx
      !!!   elsewhere
      !!!     x = 1
      !!!   end where
      !!! end if
      !!! call interaction_set_matrix_element (int, cmplx (f, kind=default))
      !!! sd = new_splitting_data (k, data%mass**2, 0._default, data%mass)
      !!! call splitting_set_t_bounds (sd, x, 1-x)
      !!! call splitting_set_collinear (sd)
      !!! q((/1, 3/)) = split_momentum (k(1), sd(1))
      !!! q((/2, 4/)) = split_momentum (k(2), sd(2))
      !!! if (all (data%affects_beam)) then
      !!!   call interaction_set_momenta (int, q(3:4), outgoing=.true.)
      !!! else if (data%affects_beam(1)) then
      !!!   call interaction_set_momenta (int, q(3:3), outgoing=.true.)
    end associate
  end subroutine

```

```

    !!! else if (data%affects_beam(2)) then
    !!!     call interaction_set_momenta (int, q(4:4), outgoing=.true.)
    !!! end if
    end associate
    sf_int%status = SF_EVALUATED
end subroutine escan_apply

```

## 10.11 Using beam event data

Instead of an analytic beam description, beam data may be provided in form of an event file. In its most simple form, the event file contains pairs of  $x$  values, relative to nominal beam energies. More advanced formats may include polarization, etc. The current implementation carries beam polarization through, if specified.

The code is very similar to the energy scan described above.

```

<sf_beam_events.f90>≡
<File header>

module sf_beam_events

  <Use kinds>
  <Use strings>
  <Use file utils>
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use flavors
  use helicities
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use sf_aux
  use sf_base

  <Standard module head>

  <SF beam events: public>

  <SF beam events: types>

contains

  <SF beam events: procedures>

end module sf_beam_events

```

### 10.11.1 Data type

```

<SF beam events: public>≡
  public :: beam_events_data_t

```



```

<SF beam events: types>≡
  type, extends(sf_data_t) :: beam_events_data_t
    private
      logical, dimension(2) :: affects_beam
      type(flavor_t), dimension(2) :: flv
      real(default), dimension(2) :: mass = 0
      type(string_t) :: file
      logical :: warn_eof = .true.
      integer :: unit = 0
    contains
      <SF beam events: beam events data: TBP>
    end type beam_events_data_t

<Beam events: public>≡
  public :: beam_events_data_init

<Beam events: procedures>≡
  subroutine beam_events_data_init (data, affects_beam, flv_in, file, warn_eof)
    type(beam_events_data_t), intent(out) :: data
    logical, dimension(2), intent(in) :: affects_beam
    type(flavor_t), dimension(2), intent(in) :: flv_in
    type(string_t), intent(in) :: file
    logical, intent(in) :: warn_eof
    data%affects_beam = affects_beam
    data%flv = flv_in
    data%mass = flavor_get_mass (data%flv)
    data%file = file
    data%warn_eof = warn_eof
  end subroutine beam_events_data_init

```

For the beam\_events SF there are no  $x$  values determined by the MC as they are directly given in the beam events file.

```

<SF beam events: beam events data: TBP>≡
  procedure :: get_n_par => beam_events_data_get_n_par

<SF beam events: procedures>≡
  function beam_events_data_get_n_par (data) result (n)
    !!! JRR: WK please check
    class(beam_events_data_t), intent(in) :: data
    integer :: n
    n = 0
  end function beam_events_data_get_n_par

<SF beam events: beam events data: TBP>+≡
  procedure :: get_pdg_out => beam_events_data_get_pdg_out

<SF beam events: procedures>+≡
  !!! JRR: WK please check
  function beam_events_data_get_pdg_out (data) result (pdg_out)
    class(beam_events_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    integer :: i, n
    n = 2
    allocate (pdg_out (n))

```

```

do i = 1, n
  pdg_out(i) = flavor_get_pdg (data%flv(i))
end do
end function beam_events_data_get_pdg_out

```

Allocate the interaction record.

```

<SF beam events: beam events data: TBP>+≡
  procedure :: allocate_sf_int => beam_events_data_allocate_sf_int

<SF beam events: procedures>+≡
  subroutine beam_events_data_allocate_sf_int (data, sf_int)
    class(beam_events_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (beam_events_t :: sf_int)
  end subroutine beam_events_data_allocate_sf_int

```

Output

```

<SF beam events: beam events data: TBP>+≡
  procedure :: write => beam_events_data_write

<SF beam events: procedures>+≡
  subroutine beam_events_data_write (data, unit, verbose) !, md5)
    class(beam_events_data_t), intent(in) :: data
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    !!! JRR: WK please check: still needed?
    ! logical, intent(in), optional :: md5
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A)") "Beam-event file data:"
    write (u, "(3x,A,A,A,A)") " prt_in = ", &
      char (flavor_get_name (data%flv(1))), &
      ", ", char (flavor_get_name (data%flv(2)))
    write (u, "(3x,A,A)") " file = '", char (data%file), "'"
    write (u, "(3x,L1)") " warn_eof = ", data%warn_eof
    write (u, "(3x,I3)") " unit = ", data%unit
  end subroutine beam_events_data_write

```

The data file needs to be opened and closed explicitly:

```

<Beam events: public>+≡
  public :: beam_events_data_open
  public :: beam_events_data_close

<Beam events: procedures>+≡
  subroutine beam_events_data_open (data)
    type(beam_events_data_t), intent(inout) :: data
    if (data%unit == 0) then
      data%unit = free_unit ()
      open (unit = data%unit, file = char (data%file))
    else
      call msg_fatal ("Reading beam events: file '" &
        // char (data%file) // "' is already open.")
    end if
    call msg_message ("Reading beam events from file '" &

```

```

        // char (data%file) // "'")
end subroutine beam_events_data_open

subroutine beam_events_data_close (data)
    type(beam_events_data_t), intent(inout) :: data
    if (data%unit /= 0) then
        close (data%unit)
        data%unit = 0
    end if
end subroutine beam_events_data_close

```

### 10.11.2 The beam events object

This is a spectrum, not a radiation. Depending on whether one or both beams are involved, we create an interaction with 1 (2) incoming and 1 (2) outgoing particles, flavor, color, and helicity being carried through. !!! JRR: WK please check I don't know actually whether this really fits into the setup done by WK.

```

<SF beam events: types>+≡
    !!! JRR: WK please check
    type, extends (sf_int_t) :: beam_events_t
        type(beam_events_data_t), pointer :: data => null ()
        real(default) :: x = 0
        real(default) :: q = 0
    contains
        <SF beam events: beam events: TBP>
    end type beam_events_t

```

Type string: show beam events file.

```

<SF beam events: beam events: TBP>≡
    procedure :: type_string => beam_events_type_string

<SF beam events: procedures>+≡
    function beam_events_type_string (object) result (string)
        !!! JRR: WK please check
        class(beam_events_t), intent(in) :: object
        type(string_t) :: string
        if (associated (object%data)) then
            string = "Beam events: " // object%data%file
        else
            string = "Beam events: [undefined]"
        end if
    end function beam_events_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF beam events: beam events: TBP>+≡
    procedure :: write => beam_events_write

<SF beam events: procedures>+≡
    subroutine beam_events_write (object, unit)
        !!! JRR: WK please check
        !!! Guess these variables do not exist for CIRCE1 (?)
        class(beam_events_t), intent(in) :: object

```

```

integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit)
if (associated (object%data)) then
  call object%data%write (u)
  if (object%status >= SF_DONE_KINEMATICS) then
    write (u, "(1x,A)") "SF parameters:"
    write (u, "(3x,A,ES17.10)") "x =", object%x
    if (object%status >= SF_FAILED_EVALUATION) then
      write (u, "(3x,A,ES17.10)") "Q =", object%q
    end if
  end if
  call object%base_write (u)
else
  write (u, "(1x,A)") "Beam events data: [undefined]"
end if
end subroutine beam_events_write

```

### 10.11.3 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

(SF beam events: beam events: TBP)+≡
  procedure :: complete_kinematics => beam_events_complete_kinematics

(SF beam events: procedures)+≡
  subroutine beam_events_complete_kinematics (sf_int, x, f, r, map)
    class(beam_events_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    logical, intent(in) :: map
    real(default) :: xb1
    if (map) then
      call msg_fatal ("Beam events: map flag not supported")
    else
      x(1) = r(1)
      f = 1
    end if
    xb1 = 1 - x(1)
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
      sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
      sf_int%x = 0
      f = 0
    end select
  end subroutine beam_events_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF beam events: beam events: TBP>+≡
  procedure :: inverse_kinematics => beam_events_inverse_kinematics

<SF beam events: procedures>+≡
  subroutine beam_events_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
    class(beam_events_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default) :: xb1
    logical :: set_mom
    set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
    if (map) then
      call msg_fatal ("Beam events: map flag not supported")
    else
      r(1) = x(1)
      f = 1
    end if
    xb1 = 1 - x(1)
    if (set_mom) then
      call sf_int%split_momentum (x, xb1)
      select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
      case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
      end select
    end if
  end subroutine beam_events_inverse_kinematics

```

```

<SF beam events: beam events: TBP>+≡
  procedure :: init => beam_events_init

<SF beam events: procedures>+≡
  subroutine beam_events_init (sf_int, data)
    class(beam_events_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(4) :: mask
    integer, dimension(4) :: hel_lock
    real(default), dimension(0) :: null_array
    type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
    type(polarization_t) :: pol1, pol2
    type(state_iterator_t) :: it_hel1, it_hel2
    integer :: i
    select type (data)
    type is (beam_events_data_t)
      if (all (data%affects_beam)) then
        hel_lock = (/ 3, 4, 1, 2 /)

```

```

        call sf_int%base_init (mask, data%mass**2, null_array, &
            data%mass**2, hel_lock = hel_lock)
    else if (data%affects_beam(1)) then
        hel_lock = (/ 2, 1, 0, 0 /)
        call sf_int%base_init (mask(1:2), [data%mass(1)**2], null_array, &
            [data%mass(1)**2], hel_lock = hel_lock(1:2))
    else if (data%affects_beam(2)) then
        hel_lock = (/ 2, 1, 0, 0 /)
        call sf_int%base_init (mask(1:2), [data%mass(2)**2], null_array, &
            [data%mass(2)**2], hel_lock = hel_lock(1:2))
    end if
    sf_int%data => data
    do i = 1, 2
        call quantum_numbers_init (qn_fc(i), &
            flv = data%flv(i), col = color_from_flavor (data%flv(i)))
        call quantum_numbers_init (qn_fc(i+2), &
            flv = data%flv(i), col = color_from_flavor (data%flv(i)))
    end do
    call polarization_init_generic (pol1, data%flv(1))
    call state_iterator_init (it_hel1, pol1%state)
    do while (state_iterator_is_valid (it_hel1))
        qn_hel(1:1) = state_iterator_get_quantum_numbers (it_hel1)
        qn_hel(3:3) = state_iterator_get_quantum_numbers (it_hel1)
        call polarization_init_generic (pol2, data%flv(2))
        call state_iterator_init (it_hel2, pol2%state)
        do while (state_iterator_is_valid (it_hel2))
            qn_hel(2:2) = state_iterator_get_quantum_numbers (it_hel2)
            qn_hel(4:4) = state_iterator_get_quantum_numbers (it_hel2)
            qn = qn_hel .merge. qn_fc
            if (all (data%affects_beam)) then
                call interaction_add_state (sf_int%interaction_t, qn)
            else if (data%affects_beam(1)) then
                call interaction_add_state (sf_int%interaction_t, qn((/1,3/)))
            else if (data%affects_beam(2)) then
                call interaction_add_state (sf_int%interaction_t, qn((/2,4/)))
            end if
            call state_iterator_advance (it_hel2)
        end do
        call polarization_final (pol2)
        call state_iterator_advance (it_hel1)
    end do
    call polarization_final (pol2)
    call interaction_freeze (sf_int%interaction_t)
    !!! JRR: WK please check
    call sf_int%set_incoming ([1,2])
    call sf_int%set_outgoing ([3,4])
    sf_int%status = SF_INITIAL
end select
end subroutine beam_events_init

```

### 10.11.4 Beam-event file structure function

This is actually not a structure function but a routine that reads a single line from the event file. If EOF is reached, we reread from the beginning and issue a warning (unless `warn_eof` is unset).

This is a rare occasion where a conditional jump is really appropriate: we want to catch EOF, but not other I/O errors.

```

<Beam events: procedures>+=
  subroutine read_x (x, data)
    real(default), dimension(:), intent(out) :: x
    type(beam_events_data_t), intent(in) :: data
    read (unit=data%unit, fmt=*, end=1) x
    return
1  if (data%warn_eof) &
    call msg_warning ("Reading beam event file: EOF reached, rewinding.")
    rewind (data%unit)
    read (unit=data%unit, fmt=*) x
  end subroutine read_x

```

### 10.11.5 Beam events application

Depending on the affected beams, we either apply the read  $x$  values to one beam or to both beams.

```

<SF beam events: beam events: TBP>+=
  procedure :: apply => beam_events_apply

<SF beam events: procedures>+=
  subroutine beam_events_apply (sf_int, scale) !, data)
    !!! JRR: WK please check
    !!! I don't understand how this works
    class(beam_events_t), intent(inout) :: sf_int
    !!! Dummy
    real(default), intent(in) :: scale
    !!! type(beam_events_data_t), intent(in) :: data
    !!! type(vector4_t), dimension(2) :: k
    !!! real(default), dimension(:), allocatable :: x
    !!! type(splitting_data_t), dimension(2) :: sd
    !!! type(vector4_t), dimension(4) :: q
    !!! complex(default), parameter :: c_one = 1
    associate (data => sf_int%data)
      !!! k(1) = interaction_get_momentum (int, 1)
      !!! if (all (data%affects_beam)) then
      !!!   k(2) = interaction_get_momentum (int, 2)
      !!! else
      !!!   k(2) = k(1)
      !!! end if
      !!! allocate (x (count (data%affects_beam)))
      !!! call read_x (x, data)
      !!! call interaction_set_matrix_element (int, c_one)
      !!! sd = new_splitting_data (k, data%mass**2, 0._default, data%mass)
      !!! call splitting_set_t_bounds (sd, x, 1-x)
      !!! call splitting_set_collinear (sd)
      !!! q((/1, 3/)) = split_momentum (k(1), sd(1))
    end associate
  end subroutine beam_events_apply

```

```

!!! q((/2, 4/)) = split_momentum (k(2), sd(2))
!!! if (all (data%affects_beam)) then
!!!   call interaction_set_momenta (int, q(3:4), outgoing=.true.)
!!! else if (data%affects_beam(1)) then
!!!   call interaction_set_momenta (int, q(3:3), outgoing=.true.)
!!! else if (data%affects_beam(2)) then
!!!   call interaction_set_momenta (int, q(4:4), outgoing=.true.)
!!! end if
end associate
sf_int%status = SF_EVALUATED
end subroutine beam_events_apply

```



## 10.12 LHAPDF

Parton distribution functions (PDFs) are available via an interface to the LHAPDF standard library.

The default PDF for protons set is chosen to be CTEQ6ll (LO fit with LO  $\alpha_s$ ).

```
<Limits: public parameters>+≡
    character(*), parameter, public :: LHAPDF_DEFAULT_PROTON = "cteq6ll.LHpdf"
    character(*), parameter, public :: LHAPDF_DEFAULT_PION   = "ABFKWPI.LHgrid"
    character(*), parameter, public :: LHAPDF_DEFAULT_PHOTON = "GSG960.LHgrid"
```

### 10.12.1 The module

```
<sf_lhapdf.f90>≡
<File header>

module sf_lhapdf

<Use kinds>
<Use strings>
    use system_dependencies, only: LHAPDF_PDFSETS_PATH !NODEP!
    use system_dependencies, only: LHAPDF_AVAILABLE !NODEP!
    use limits, only: LHAPDF_DEFAULT_PROTON !NODEP!
    use limits, only: LHAPDF_DEFAULT_PION !NODEP!
    use limits, only: LHAPDF_DEFAULT_PHOTON !NODEP!
<Use file utils>
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base

<Standard module head>

<SF lhpdf: public>

<SF lhpdf: types>

<SF lhpdf: interfaces>

contains

<SF lhpdf: procedures>

<SF lhpdf: tests>
```

```
end module sf_lhapdf
```

### 10.12.2 LHAPDF library interface

Here we specify explicit interfaces for all LHAPDF routines that we use below.

$\langle SF \text{ lhpdf: interfaces} \rangle \equiv$

```
interface
  subroutine InitPDFsetM (set, file)
    integer, intent(in) :: set
    character(*), intent(in) :: file
  end subroutine InitPDFsetM
end interface
```

$\langle SF \text{ lhpdf: interfaces} \rangle + \equiv$

```
interface
  subroutine InitPDFM (set, mem)
    integer, intent(in) :: set, mem
  end subroutine InitPDFM
end interface
```

$\langle SF \text{ lhpdf: interfaces} \rangle + \equiv$

```
interface
  subroutine numberPDFM (set, n_members)
    integer, intent(in) :: set
    integer, intent(out) :: n_members
  end subroutine numberPDFM
end interface
```

$\langle SF \text{ lhpdf: interfaces} \rangle + \equiv$

```
interface
  subroutine evolvePDFM (set, x, q, ff)
    integer, intent(in) :: set
    double precision, intent(in) :: x, q
    double precision, dimension(-6:6), intent(out) :: ff
  end subroutine evolvePDFM
end interface
```

$\langle SF \text{ lhpdf: interfaces} \rangle + \equiv$

```
interface
  subroutine evolvePDFphotonM (set, x, q, ff, fphot)
    integer, intent(in) :: set
    double precision, intent(in) :: x, q
    double precision, dimension(-6:6), intent(out) :: ff
    double precision, intent(out) :: fphot
  end subroutine evolvePDFphotonM
end interface
```

$\langle SF \text{ lhpdf: interfaces} \rangle + \equiv$

```
interface
  subroutine evolvePDFpM (set, x, q, s, scheme, ff)
```

```

        integer, intent(in) :: set
        double precision, intent(in) :: x, q, s
        integer, intent(in) :: scheme
        double precision, dimension(-6:6), intent(out) :: ff
    end subroutine evolvePDFpM
end interface

<SF lhpdf: interfaces>+=
interface
    subroutine GetXminM (set, mem, xmin)
        integer, intent(in) :: set, mem
        double precision, intent(out) :: xmin
    end subroutine GetXminM
end interface

<SF lhpdf: interfaces>+=
interface
    subroutine GetXmaxM (set, mem, xmax)
        integer, intent(in) :: set, mem
        double precision, intent(out) :: xmax
    end subroutine GetXmaxM
end interface

<SF lhpdf: interfaces>+=
interface
    subroutine GetQ2minM (set, mem, q2min)
        integer, intent(in) :: set, mem
        double precision, intent(out) :: q2min
    end subroutine GetQ2minM
end interface

<SF lhpdf: interfaces>+=
interface
    subroutine GetQ2maxM (set, mem, q2max)
        integer, intent(in) :: set, mem
        double precision, intent(out) :: q2max
    end subroutine GetQ2maxM
end interface

<SF lhpdf: interfaces>+=
interface
    function has_photon () result(flag)
        logical :: flag
    end function has_photon
end interface

```

### 10.12.3 The LHAPDF status

This type holds the initialization status of the LHAPDF system. Entry 1 is for proton PDFs, entry 2 for pion PDFs, entry 3 for photon PDFs.

```
<SF lhpdf: public>=
```

```

    public :: lhpdf_status_t
<SF lhpdf: types>≡
    type :: lhpdf_status_t
        private
            logical, dimension(3) :: initialized = .false.
    end type lhpdf_status_t

<SF lhpdf: public>+≡
    public :: lhpdf_status_reset

<SF lhpdf: procedures>≡
    subroutine lhpdf_status_reset (lhpdf_status)
        type(lhpdf_status_t), intent(inout) :: lhpdf_status
        lhpdf_status%initialized = .false.
    end subroutine lhpdf_status_reset

<SF lhpdf: procedures>+≡
    function lhpdf_status_is_initialized (lhpdf_status, set) result (flag)
        logical :: flag
        type(lhpdf_status_t), intent(in) :: lhpdf_status
        integer, intent(in), optional :: set
        if (present (set)) then
            select case (set)
            case (1:3);    flag = lhpdf_status%initialized(set)
            case default;  flag = .false.
            end select
        else
            flag = any (lhpdf_status%initialized)
        end if
    end function lhpdf_status_is_initialized

<SF lhpdf: procedures>+≡
    subroutine lhpdf_status_set_initialized (lhpdf_status, set)
        type(lhpdf_status_t), intent(inout) :: lhpdf_status
        integer, intent(in) :: set
        lhpdf_status%initialized(set) = .true.
    end subroutine lhpdf_status_set_initialized

```

#### 10.12.4 LHAPDF initialization

Before using LHAPDF, we have to initialize it with a particular data set and member. This applies not just if we use structure functions, but also if we just use an  $\alpha_s$  formula. The integer `set` should be 1 for proton, 2 for pion, and 3 for photon, but this is just convention.

If the particular set has already been initialized, do nothing. This implies that whenever we want to change the setup for a particular set, we have to reset the LHAPDF status. `lhpdf_initialize` has an obvious name clash with `lhpdf_init`, the reason it works for `pdf_builtin` is that these things are outsourced to a separate module (inc. `lhpdf_status` etc.).

```

<SF lhpdf: public>+≡
    public :: lhpdf_initialize

```

```

<SF lhpdf: procedures>+≡
subroutine lhpdf_initialize (status, set, prefix, file, member)
  type(lhpdf_status_t), intent(inout) :: status
  integer, intent(in) :: set
  type(string_t), intent(inout) :: prefix
  type(string_t), intent(inout) :: file
  integer, intent(inout) :: member
  if (lhpdf_status_is_initialized (status, set)) return
  if (prefix == "") prefix = LHAPDF_PDFSETS_PATH
  if (file == "") then
    select case (set)
      case (1); file = LHAPDF_DEFAULT_PROTON
      case (2); file = LHAPDF_DEFAULT_PION
      case (3); file = LHAPDF_DEFAULT_PHOTON
    end select
  end if
  if (data_file_exists (prefix // "/" // file)) then
    call InitPDFsetM (set, char (prefix // "/" // file))
  else
    call msg_fatal ("LHAPDF: Data file '" &
      // char (file) // "' not found in '" // char (prefix) // "'")
    return
  end if
  if (.not. dataset_member_exists (set, member)) then
    call msg_error (" LHAPDF: Chosen member does not exist for set '" &
      // char (file) // "', using default.")
    member = 0
  end if
  call InitPDFM (set, member)
  call lhpdf_status_set_initialized (status, set)
contains
  function data_file_exists (fq_name) result (exist)
    type(string_t), intent(in) :: fq_name
    logical :: exist
    inquire (file = char(fq_name), exist = exist)
  end function data_file_exists
  function dataset_member_exists (set, member) result (exist)
    integer, intent(in) :: set, member
    logical :: exist
    integer :: n_members
    call numberPDFM (set, n_members)
    exist = member >= 0 .and. member <= n_members
  end function dataset_member_exists
end subroutine lhpdf_initialize

```

### 10.12.5 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF lhpdf: lhpdf: TBP>≡

```

```

    procedure :: complete_kinematics => lhpdf_complete_kinematics
<SF lhpdf: procedures>+≡
    subroutine lhpdf_complete_kinematics (sf_int, x, f, r, map)
        class(lhpdf_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(out) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: r
        logical, intent(in) :: map
        real(default) :: xb1
        if (map) then
            call msg_fatal ("LHAPDF: map flag not supported")
        else
            x(1) = r(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        call sf_int%split_momentum (x, xb1)
        select case (sf_int%status)
        case (SF_DONE_KINEMATICS)
            sf_int%x = x(1)
        case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
        end select
    end subroutine lhpdf_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF lhpdf: lhpdf: TBP>+≡
    procedure :: inverse_kinematics => lhpdf_inverse_kinematics
<SF lhpdf: procedures>+≡
    subroutine lhpdf_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        class(lhpdf_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
        real(default) :: xb1
        logical :: set_mom
        set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
        if (map) then
            call msg_fatal ("LHAPDF: map flag not supported")
        else
            r(1) = x(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        if (set_mom) then
            call sf_int%split_momentum (x, xb1)
            select case (sf_int%status)

```

```

        case (SF_DONE_KINEMATICS)
            sf_int%x = x(1)
        case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
        end select
    end if
end subroutine lhpdf_inverse_kinematics

```

### 10.12.6 The LHAPDF data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

```

<SF lhpdf: public>+≡
    public :: lhpdf_data_t

<SF lhpdf: types>+≡
    type, extends (sf_data_t) :: lhpdf_data_t
    private
        type(string_t) :: prefix
        type(string_t) :: file
        integer :: member = 0
        type(model_t), pointer :: model => null ()
        type(flavor_t) :: flv_in
        integer :: set = 0
        logical :: invert = .false.
        logical :: photon = .false.
        logical :: has_photon = .false.
        integer :: photon_scheme = 0
        real(default) :: xmin = 0, xmax = 0
        real(default) :: qmin = 0, qmax = 0
        logical, dimension(-6:6) :: mask = .true.
        logical :: mask_photon = .true.
    contains
        <SF lhpdf: lhpdf data: TBP>
    end type lhpdf_data_t

```

Generate PDF data. This is provided as a function, but it has the side-effect of initializing the requested PDF set. A finalizer is not needed.

The library uses double precision, so since the default precision may be quadruple, we use auxiliary variables for type casting.

```

<SF lhpdf: lhpdf data: TBP>≡
    procedure :: init => lhpdf_data_init

<SF lhpdf: procedures>+≡
    subroutine lhpdf_data_init &
        (data, status, model, flv, prefix, file, member, photon_scheme)
        class(lhpdf_data_t), intent(out) :: data
        type(lhpdf_status_t), intent(inout) :: status
    end subroutine

```

```

type(model_t), intent(in), target :: model
type(flavor_t), intent(in) :: flv
type(string_t), intent(in), optional :: prefix, file
integer, intent(in), optional :: member
integer, intent(in), optional :: photon_scheme
integer :: mem
double precision :: xmin, xmax, q2min, q2max
external :: InitPDFsetM, InitPDFM, numberPDFM
external :: GetXminM, GetXmaxM, GetQ2minM, GetQ2maxM
if (.not. LHAPDF_AVAILABLE) then
    call msg_fatal ("LHAPDF requested but library is not linked")
    return
end if
data%model => model
data%flv_in = flv
select case (flavor_get_pdg (flv))
case (PROTON)
    data%set = 1
case (-PROTON)
    data%set = 1
    data%invert = .true.
case (PIPLUS)
    data%set = 2
case (-PIPLUS)
    data%set = 2
    data%invert = .true.
case (PHOTON)
    data%set = 3
    data%photon = .true.
    if (present (photon_scheme)) data%photon_scheme = photon_scheme
case default
    call msg_fatal (" LHAPDF: " &
        // "incoming particle must be (anti)proton, pion, or photon.")
    return
end select
if (present (prefix)) then
    data%prefix = prefix
else
    data%prefix = ""
end if
if (present (file)) then
    data%file = file
else
    data%file = ""
end if
call lhpdf_initialize &
    (status, data%set, data%prefix, data%file, data%member)
call GetXminM (data%set, data%member, xmin)
call GetXmaxM (data%set, data%member, xmax)
call GetQ2minM (data%set, data%member, q2min)
call GetQ2maxM (data%set, data%member, q2max)
data%xmin = xmin
data%xmax = xmax
data%qmin = sqrt (q2min)

```



```

        data%qmax = sqrt (q2max)
        data%has_photon = has_photon ()
    end subroutine lhpdf_data_init

```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF will generate the corresponding flavor on output.

```

<LHAPDF: public>≡
    public :: lhpdf_data_set_mask

<LHAPDF: procedures>≡
    subroutine lhpdf_data_set_mask (data, mask)
        type(lhpdf_data_t), intent(inout) :: data
        logical, dimension(-6:6), intent(in) :: mask
        data%mask = mask
    end subroutine lhpdf_data_set_mask

```

Return the public part of the data set.

```

<LHAPDF: public>+≡
    public :: lhpdf_data_get_public_info

<LHAPDF: procedures>+≡
    subroutine lhpdf_data_get_public_info &
        (data, lhpdf_dir, lhpdf_file, lhpdf_member)
        type(lhpdf_data_t), intent(in) :: data
        type(string_t), intent(out) :: lhpdf_dir, lhpdf_file
        integer, intent(out) :: lhpdf_member
        lhpdf_dir = data%prefix
        lhpdf_file = data%file
        lhpdf_member = data%member
    end subroutine lhpdf_data_get_public_info

```

Return the number of the member of the data set.

```

<LHAPDF: public>+≡
    public :: lhpdf_data_get_set

<LHAPDF: procedures>+≡
    function lhpdf_data_get_set(data) result(set)
        type(lhpdf_data_t), intent(in) :: data
        integer :: set
        set = data%set
    end function lhpdf_data_get_set

```

Output

```

<SF lhpdf: lhpdf data: TBP>+≡
    procedure :: write => lhpdf_data_write

<SF lhpdf: procedures>+≡
    subroutine lhpdf_data_write (data, unit, verbose) !, md5, beam_fmt)
        class(lhpdf_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        logical :: verb
        integer :: u

```

```

!!! JRR: WK please check
!!! Do we still need this? seems it wasn't needed in 2.1.1. either
if (present (verbose)) then
    verb = verbose
else
    verb = .false.
end if
! logical, intent(in), optional :: md5
! logical, intent(in), optional :: beam_fmt
! logical :: is_md5, is_beam_fmt
! if (present (md5)) then
!     is_md5 = md5
! else
!     is_md5 = .false.
! end if
! if (present (beam_fmt)) then
!     is_beam_fmt = beam_fmt
! else
!     is_beam_fmt = .false.
! end if
u = output_unit (unit); if (u < 0) return
write (u, "(1x,A)") "LHAPDF data:"
if (data%set /= 0) then
    write (u, "(3x,A)", advance="no") "flavor          = "
    call flavor_write (data%flv_in, u); write (u, *)
    !!! JRR: WK please check: Do we still need this?
    ! if (.not. is_md5) &
    if (verb) then
        write (u, "(3x,A,A)"      " prefix          = ", char (data%prefix)
    else
        write (u, "(3x,A,A)"      " prefix          = ", &
            " <empty (non-verbose version)>"
    end if
    write (u, "(3x,A,A)"      " file              = ", char (data%file)
    write (u, "(3x,A,I3)"     " member           = ", data%member
    write (u, "(3x,A,ES19.12)" " x(min)            = ", data%xmin
    write (u, "(3x,A,ES19.12)" " x(max)           = ", data%xmax
    write (u, "(3x,A,ES19.12)" " Q^2(min)        = ", data%qmin
    write (u, "(3x,A,ES19.12)" " Q^2(max)        = ", data%qmax
    write (u, "(3x,A,L1)"     " invert          = ", data%invert
    if (data%photon) write (u, "(3x,A,I3)" &
        " IP2 (scheme) = ", data%photon_scheme
    !!! JRR: WK please check: still needed?
    ! if (.not. is_beam_fmt) then
        write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))" &
            " mask              = ", &
            data%mask(-6:-1), " ", data%mask(0), " ", data%mask(1:6)
        write (u, "(3x,A,L1)"      " photon mask = ", data%mask_photon
    ! end if
else
    write (u, "(3x,A)") "[undefined]"
end if
end subroutine lhpdf_data_write

```

The number of parameters is one. We do not generate transverse momentum.

```

<SF lhpdf: lhpdf data: TBP>+=
  procedure :: get_n_par => lhpdf_data_get_n_par

<SF lhpdf: procedures>+=
  function lhpdf_data_get_n_par (data) result (n)
    class(lhpdf_data_t), intent(in) :: data
    integer :: n
    n = 1
  end function lhpdf_data_get_n_par

```

Return the outgoing particle PDG codes. This is based on the mask.

```

<SF lhpdf: lhpdf data: TBP>+=
  procedure :: get_pdg_out => lhpdf_data_get_pdg_out

<SF lhpdf: procedures>+=
  function lhpdf_data_get_pdg_out (data) result (pdg_out)
    class(lhpdf_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    integer :: n, np, i
    n = count (data%mask)
    np = 0; if (data%has_photon .and. data%mask_photon) np = 1
    allocate (pdg_out (n + np))
    pdg_out(1:n) = pack ([i, i = -6, 6], data%mask)
    if (np == 1) pdg_out(n+np) = PHOTON
  end function lhpdf_data_get_pdg_out

```

Allocate the interaction record.

```

<SF lhpdf: lhpdf data: TBP>+=
  procedure :: allocate_sf_int => lhpdf_data_allocate_sf_int

<SF lhpdf: procedures>+=
  subroutine lhpdf_data_allocate_sf_int (data, sf_int)
    class(lhpdf_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (lhpdf_t :: sf_int)
  end subroutine lhpdf_data_allocate_sf_int

```

## 10.12.7 The LHAPDF object

The `lhpdf_t` data type is a  $1 \rightarrow 2$  interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

This is the LHAPDF object which holds input data together with the interaction. We also store the  $x$  momentum fraction and the scale, since kinematics and function value are requested at different times.

The PDF application is a  $1 \rightarrow 2$  splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

```

<SF lhpdf: types>+≡
type, extends (sf_int_t) :: lhpdf_t
  type(lhpdf_data_t), pointer :: data => null ()
  real(default) :: x = 0
  real(default) :: q = 0
  real(default) :: s = 0
contains
  <SF lhpdf: lhpdf: TBP>
end type lhpdf_t

```

Type string: display the chosen PDF set.

```

<SF lhpdf: lhpdf: TBP>+≡
  procedure :: type_string => lhpdf_type_string

<SF lhpdf: procedures>+≡
function lhpdf_type_string (object) result (string)
  class(lhpdf_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
    string = "LHAPDF: " // object%data%file
  else
    string = "LHAPDF: [undefined]"
  end if
end function lhpdf_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF lhpdf: lhpdf: TBP>+≡
  procedure :: write => lhpdf_write

<SF lhpdf: procedures>+≡
subroutine lhpdf_write (object, unit)
  class(lhpdf_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
    call object%data%write (u)
    if (object%status >= SF_DONE_KINEMATICS) then
      write (u, "(1x,A)") "SF parameters:"
      write (u, "(3x,A,ES17.10)") "x =", object%x
      if (object%status >= SF_FAILED_EVALUATION) then
        write (u, "(3x,A,ES17.10)") "Q =", object%q
      end if
    end if
    call object%base_write (u)
  else
    write (u, "(1x,A)") "LHAPDF data: [undefined]"
  end if
end subroutine lhpdf_write

```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

```

<SF lhpdf: lhpdf: TBP>+≡
  procedure :: init => lhpdf_init

<SF lhpdf: procedures>+≡
  subroutine lhpdf_init (sf_int, data)
    class(lhpdf_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    type(flavor_t) :: flv, flv_remnant
    type(quantum_numbers_t), dimension(3) :: qn
    integer :: i
    select type (data)
    type is (lhpdf_data_t)
      mask = new_quantum_numbers_mask (.false., .false., .true.)
      call sf_int%base_init (mask, [0._default], [0._default], [0._default])
      sf_int%data => data
      do i = -6, 6
        if (data%mask(i)) then
          call quantum_numbers_init (qn(1), data%flv_in)
          if (i == 0) then
            call flavor_init (flv, GLUON, data%model)
            call flavor_init (flv_remnant, HADRON_REMNANT_OCTET, data%model)
          else
            call flavor_init (flv, i, data%model)
            call flavor_init (flv_remnant, &
              sign (HADRON_REMNANT_TRIPLET, -i), data%model)
          end if
          call quantum_numbers_init (qn(2), &
            flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
          call quantum_numbers_init (qn(3), &
            flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
          call interaction_add_state (sf_int%interaction_t, qn)
        end if
      end do
      if (data%has_photon .and. data%mask_photon) then
        call flavor_init (flv, PHOTON, data%model)
        call flavor_init (flv_remnant, HADRON_REMNANT_SINGLET, data%model)
        call quantum_numbers_init (qn(2), flv = flv_remnant, &
          col = color_from_flavor (flv_remnant, 1))
        call quantum_numbers_init (qn(3), flv = flv, &
          col = color_from_flavor (flv, 1, reverse=.true.))
        call interaction_add_state (sf_int%interaction_t, qn)
      end if
      call interaction_freeze (sf_int%interaction_t)
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
      sf_int%status = SF_INITIAL
    end select
  end subroutine lhpdf_init

```

### 10.12.8 Structure function

For the PDFs, we separate kinematics from dynamics. We first generate appropriate  $x$  values, independent of flavor and scale. This allows us to determine the momenta that initiate the hard scattering. Only when the hard process has been computed, we can be sure about the scattering scale and compute the structure function values.

For the  $x$  values, we can only apply a simple mapping that eliminates the  $1/x$  singularity, i.e., we generate  $x$  on a logarithmic scale. This produces a Jacobian factor that we store along with the generated  $x$  value. The boundaries are used as returned by the LHAPDF library for the chosen PDF set.

```

(LHAPDF: procedures)+≡
  subroutine generate_x (x, f, r, lhpdf_data)
    real(default), intent(out) :: x, f
    real(default), intent(in)  :: r
    type(lhpdf_data_t), intent(in) :: lhpdf_data
    real(default) :: lg
    x = r
    f = 1
    !   lg = log (lhpdf_data% xmax / lhpdf_data% xmin)
    !   x = lhpdf_data% xmin * exp (r * lg)
    !   f = x * lg
  end subroutine generate_x

```

The previous routine is called here, so we can compute the complete kinematics.

(There is no mapping involved [see above], otherwise we should take and respect a `no_map` flag.)

```

(LHAPDF: public)+≡
  public :: interaction_set_kinematics_lhpdf

(LHAPDF: procedures)+≡
  subroutine interaction_set_kinematics_lhpdf (int, x, f, s, r, lhpdf_data)
    type(interaction_t), intent(inout) :: int
    real(default), intent(out) :: x, f, s
    real(default), intent(in)  :: r
    type(lhpdf_data_t), intent(in) :: lhpdf_data
    type(vector4_t) :: k
    type(splitting_data_t) :: sd
    call generate_x (x, f, r, lhpdf_data)
    k = interaction_get_momentum (int, 1)
    s = k**2
    sd = new_splitting_data (k, s, 0._default, 0._default)
    call splitting_set_t_bounds (sd, x, 1 - x)
    call splitting_set_collinear (sd)
    call interaction_set_momenta &
      (int, split_momentum (k, sd), outgoing=.true.)
  end subroutine interaction_set_kinematics_lhpdf

```

Once the scale is also known, we can actually call the library and set the values. If the scale is out of bounds, we reset it to the boundary. Account for the Jacobian.

We have to cast the LHAPDF arguments to/from double precision (possibly from/to quadruple precision), if necessary. Furthermore, some structure functions can yield negative results (sea quarks close to  $x = 1$ ). We set these unphysical values to zero.

```

<SF lhpdf: lhpdf: TBP>+=
  procedure :: apply => lhpdf_apply

<SF lhpdf: procedures>+=
  subroutine lhpdf_apply (sf_int, scale)
    class(lhapdf_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default) :: x, s
    double precision :: xx, qq, ss
    double precision, dimension(-6:6) :: ff
    double precision :: fphot
    complex(default), dimension(:), allocatable :: fc
    external :: evolvePDFM, evolvePDFpM
    associate (data => sf_int%data)
      sf_int%q = scale
      x = sf_int%x
      s = sf_int%s
      qq = min (data% qmax, scale)
      qq = max (data% qmin, qq)
      if (.not. data% photon) then
        xx = x
        if (data% invert) then
          if (data% has_photon) then
            call evolvePDFphotonM (data% set, xx, qq, ff(6:-6:-1), fphot)
          else
            call evolvePDFM (data% set, xx, qq, ff(6:-6:-1))
          end if
        else
          if (data% has_photon) then
            call evolvePDFphotonM (data% set, xx, qq, ff, fphot)
          else
            call evolvePDFM (data% set, xx, qq, ff)
          end if
        end if
      else
        ss = s
        call evolvePDFpM (data% set, xx, qq, &
          ss, data% photon_scheme, ff)
      end if
      if (data% has_photon) then
        allocate (fc (count ((/data%mask, data%mask_photon/))))
        fc = max (pack ((/ff, fphot/) / x, &
          (/data% mask, data%mask_photon/)), 0._default)
      else
        allocate (fc (count (data%mask)))
        fc = max (pack (ff / x, data%mask), 0._default)
      end if
    end associate
  end subroutine

```

```

end associate
call interaction_set_matrix_element (sf_int%interaction_t, fc)
sf_int%status = SF_EVALUATED
end subroutine lhpdf_apply

```

*<CCC SF lhpdf: procedures>≡*

```

!! !! subroutine interaction_apply_lhpdf (int, scale, x, f, s, lhpdf_data)
!! !!   type(interaction_t), intent(inout) :: int
!! !!   real(default), intent(in) :: scale, x, f, s
!! !!   type(lhpdf_data_t), intent(in) :: lhpdf_data
!! !!   double precision :: xx, qq, ss
!! !!   double precision, dimension(-6:6) :: ff
!! !!   double precision :: fphot
!! !!   complex(default), dimension(:), allocatable :: fc
!! !!   external :: evolvePDFM, evolvePDFpM
!! !!   xx = x
!! !!   qq = min (lhpdf_data% qmax, scale)
!! !!   qq = max (lhpdf_data% qmin, qq)
!! !!   if (.not. lhpdf_data% photon) then
!! !!     if (lhpdf_data% invert) then
!! !!       if (lhpdf_data%has_photon) then
!! !!         call evolvePDFphotonM (lhpdf_data% set, xx, qq, ff(6:-6:-1), fphot)
!! !!       else
!! !!         call evolvePDFM (lhpdf_data% set, xx, qq, ff(6:-6:-1))
!! !!       end if
!! !!     else
!! !!       if (lhpdf_data%has_photon) then
!! !!         call evolvePDFphotonM (lhpdf_data% set, xx, qq, ff, fphot)
!! !!       else
!! !!         call evolvePDFM (lhpdf_data% set, xx, qq, ff)
!! !!       end if
!! !!     end if
!! !!   else
!! !!     ss = s
!! !!     call evolvePDFpM (lhpdf_data% set, xx, qq, &
!! !!       ss, lhpdf_data% photon_scheme, ff)
!! !!   end if
!! !!   if (lhpdf_data%has_photon) then
!! !!     allocate (fc (count ((/lhpdf_data%mask, lhpdf_data%mask_photon/))))
!! !!     fc = max (pack ((/ff, fphot/) / x, &
!! !!       (/lhpdf_data% mask, lhpdf_data%mask_photon/)) * f, 0._default)
!! !!   else
!! !!     allocate (fc (count (lhpdf_data%mask)))
!! !!     fc = max (pack (ff / x, lhpdf_data%mask) * f, 0._default)
!! !!   end if
!! !!   call interaction_set_matrix_element (int, fc)
!! !! end subroutine interaction_apply_lhpdf

```

### 10.12.9 Unit tests

*<SF lhpdf: public>+≡*

```

public :: sf_lhpdf_test

```



```

<SF lhpdf: tests>≡
  subroutine sf_lhapdf_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <SF lhpdf: execute tests>
  end subroutine sf_lhapdf_test

```

## Test structure function data

Construct and display a test structure function data object.

```

<SF lhpdf: execute tests>≡
  call test (sf_lhapdf_1, "sf_lhapdf_1", &
    "structure function configuration", &
    u, results)

<SF lhpdf: tests>+≡
  subroutine sf_lhapdf_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable :: data
    type(lhapdf_status_t) :: status

    write (u, "(A)")  "* Test output: sf_lhapdf_1"
    write (u, "(A)")  "* Purpose: initialize and display &
      &test structure function data"
    write (u, "(A)")

    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("QCD"), &
      var_str ("QCD.mdl"), os_data, model)
    call flavor_init (flv, PROTON, model)

    allocate (lhpdf_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    select type (data)
    type is (lhpdf_data_t)
      call data%init (status, model, flv)
    end select

    call data%write (u)

    write (u, "(A)")

```

```

write (u, "(1x,A)") "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))") data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_lhapdf_1"

end subroutine sf_lhapdf_1

```

### Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

```

<SF lhpdf: execute tests>+≡
call test (sf_lhapdf_2, "sf_lhapdf_2", &
  "structure function instance", &
  u, results)

<SF lhpdf: tests>+≡
subroutine sf_lhapdf_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(lhapdf_status_t) :: status
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, x
  real(default) :: f, s

  write (u, "(A)")  "* Test output: sf_lhapdf_2"
  write (u, "(A)")  "* Purpose: initialize and fill &
    &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("QCD"), &
    var_str ("QCD.mdl"), os_data, model)
  call flavor_init (flv, PROTON, model)

  call reset_interaction_counter ()

```

```

allocate (lhpdf_data_t :: data)
select type (data)
type is (lhpdf_data_t)
    call data%init (status, model, flv)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0.5_default
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

```

```

write (u, "(A,9(1x,F12.9))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate for Q = 100 GeV"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_lhapdf_2"

end subroutine sf_lhapdf_2

```

## 10.13 Builtin PDF sets

For convenience as well and to provide functionality not covered by LHAPDF (like photons as partons), we ship some PDFs with WHIZARD.

```

<Limits: public parameters>+≡
character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PROTON = "CTEQ6L"
character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PION  = "NONE"
character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PHOTON = "NONE"

```

### 10.13.1 The module

```

(sf_pdf_builtin.f90)≡
<File header>

module sf_pdf_builtin

  <Use kinds>
  <Use strings>
  use limits, only: PDF_BUILTIN_DEFAULT_PROTON !NODEP!
  use limits, only: PDF_BUILTIN_DEFAULT_PION  !NODEP!
  use limits, only: PDF_BUILTIN_DEFAULT_PHOTON !NODEP!
  <Use file utils>
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use pdf_builtin !NODEP!
  use unit_tests
  use os_interface
  use sm_qcd
  use models

```

```

    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base

    <Standard module head>

    <SF pdf builtin: public>

    <SF pdf builtin: types>

    contains

    <SF pdf builtin: procedures>

    <SF pdf builtin: tests>

    end module sf_pdf_builtin

```

### 10.13.2 The PDF builtin data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

```

<SF pdf builtin: public>≡
    public :: pdf_builtin_data_t

<SF pdf builtin: types>≡
    type, extends (sf_data_t) :: pdf_builtin_data_t
    private
    integer :: id = -1
    type (string_t) :: name
    type(model_t), pointer :: model => null ()
    type(flavor_t) :: flv_in
    logical :: invert
    logical :: has_photon
    logical :: photon
    logical, dimension(-6:6) :: mask
    logical :: mask_photon
    contains
    <SF pdf builtin: pdf builtin data: TBP>
    end type pdf_builtin_data_t

```

Generate PDF data and initialize the requested set. Pion and photon PDFs are disabled at the moment until we ship appropriate structure functions. needed.

```

<SF pdf builtin: pdf builtin data: TBP>≡
    procedure :: init => pdf_builtin_data_init

```

```

<SF pdf builtin: procedures>≡
subroutine pdf_builtin_data_init (data, pdf_status, model, flv, name, path)
  class(pdf_builtin_data_t), intent(out) :: data
  type(pdf_builtin_status_t), intent(inout) :: pdf_status
  type(model_t), intent(in), target :: model
  type(flavor_t), intent(in) :: flv
  type(string_t), intent(in) :: name
  type(string_t), intent(in) :: path
  data%model => model
  data%flv_in = flv
  data%mask = .true.
  data%mask_photon = .true.
  select case (flavor_get_pdg (flv))
  case (PROTON)
    data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
    data%invert = .false.
    data%photon = .false.
  case (-PROTON)
    data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
    data%invert = .true.
    data%photon = .false.
  ! case (PIPLUS)
  !   data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
  !   data%invert = .false.
  !   data%photon = .false.
  ! case (-PIPLUS)
  !   data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
  !   data%invert = .true.
  !   data%photon = .false.
  ! case (PHOTON)
  !   data%name = var_str (PDF_BUILTIN_DEFAULT_PHOTON)
  !   data%invert = .false.
  !   data%photon = .true.
  case default
    call msg_fatal (" PDF: " &
      // "incoming particle must either proton or antiproton.")
    return
  end select
  data%name = name
  data%id = pdf_get_id (data%name)
  if (data%id < 0) call msg_fatal ("unknown PDF set " // char (data%name))
  data%has_photon = pdf_provides_photon (data%id)
  call pdf_init (pdf_status, data%id, path)
end subroutine pdf_builtin_data_init

```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF will generate the corresponding flavor on output.

```

<CCC SF pdf builtin: public>≡
  public :: pdf_builtin_data_set_mask

<CCC SF pdf builtin: procedures>≡
subroutine pdf_builtin_data_set_mask (data, mask)
  type(pdf_builtin_data_t), intent(inout) :: data
  logical, dimension(-6:6), intent(in) :: mask

```

```

    data%mask = mask
end subroutine pdf_builtin_data_set_mask

```

Output.

```

<SF pdf builtin: pdf builtin data: TBP>+≡
  procedure :: write => pdf_builtin_data_write

<SF pdf builtin: procedures>+≡
  subroutine pdf_builtin_data_write (data, unit, verbose)
    class(pdf_builtin_data_t), intent(in) :: data
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A)") "PDF builtin data:"
    if (data%id < 0) then
      write (u, "(3x,A)") "[undefined]"
      return
    end if
    write (u, "(3x,A)", advance="no") "flavor      = "
    call flavor_write (data%flv_in, u); write (u, *)
    write (u, "(3x,A,A)") "name          = ", char (data%name)
    write (u, "(3x,A,L1)") "invert       = ", data%invert
    write (u, "(3x,A,L1)") "has photon   = ", data%has_photon
    write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))") &
      "mask      = ", &
      data%mask(-6:-1), " ", data%mask(0), " ", data%mask(1:6)
    write (u, "(3x,A,L1)") "photon mask = ", data%mask_photon
  end subroutine pdf_builtin_data_write

```

The number of parameters is one. We do not generate transverse momentum.

```

<SF pdf builtin: pdf builtin data: TBP>+≡
  procedure :: get_n_par => pdf_builtin_data_get_n_par

<SF pdf builtin: procedures>+≡
  function pdf_builtin_data_get_n_par (data) result (n)
    class(pdf_builtin_data_t), intent(in) :: data
    integer :: n
    n = 1
  end function pdf_builtin_data_get_n_par

```

Return the outgoing particle PDG codes. This is based on the mask.

```

<SF pdf builtin: pdf builtin data: TBP>+≡
  procedure :: get_pdg_out => pdf_builtin_data_get_pdg_out

<SF pdf builtin: procedures>+≡
  function pdf_builtin_data_get_pdg_out (data) result (pdg_out)
    class(pdf_builtin_data_t), intent(in) :: data
    integer, dimension(:), allocatable :: pdg_out
    integer :: n, np, i
    n = count (data%mask)
    np = 0; if (data%has_photon .and. data%mask_photon) np = 1
    allocate (pdg_out (n + np))
    pdg_out(1:n) = pack ([i, i = -6, 6], data%mask)

```

```

    if (np == 1) pdg_out(n+np) = PHOTON
end function pdf_builtin_data_get_pdg_out

```

Allocate the interaction record.

```

⟨SF pdf builtin: pdf builtin data: TBP⟩+≡
    procedure :: allocate_sf_int => pdf_builtin_data_allocate_sf_int

⟨SF pdf builtin: procedures⟩+≡
    subroutine pdf_builtin_data_allocate_sf_int (data, sf_int)
        class(pdf_builtin_data_t), intent(in) :: data
        class(sf_int_t), intent(inout), allocatable :: sf_int
        allocate (pdf_builtin_t :: sf_int)
    end subroutine pdf_builtin_data_allocate_sf_int

```

### 10.13.3 The PDF object

The PDF  $1 \rightarrow 2$  interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

The PDF application is a  $1 \rightarrow 2$  splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

```

⟨SF pdf builtin: types⟩+≡
    type, extends (sf_int_t) :: pdf_builtin_t
        type(pdf_builtin_data_t), pointer :: data => null ()
        real(default) :: x = 0
        real(default) :: q = 0
    contains
        ⟨SF pdf builtin: pdf builtin: TBP⟩
    end type pdf_builtin_t

```

Type string: display the chosen PDF set.

```

⟨SF pdf builtin: pdf builtin: TBP⟩≡
    procedure :: type_string => pdf_builtin_type_string

⟨SF pdf builtin: procedures⟩+≡
    function pdf_builtin_type_string (object) result (string)
        class(pdf_builtin_t), intent(in) :: object
        type(string_t) :: string
        if (associated (object%data)) then
            string = "PDF builtin: " // object%data%name
        else
            string = "PDF builtin: [undefined]"
        end if
    end function pdf_builtin_type_string

```



Output. Call the interaction routine after displaying the configuration.

```

<SF pdf builtin: pdf builtin: TBP>+≡
  procedure :: write => pdf_builtin_write

<SF pdf builtin: procedures>+≡
  subroutine pdf_builtin_write (object, unit)
    class(pdf_builtin_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    if (associated (object%data)) then
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)") "SF parameters:"
        write (u, "(3x,A,ES17.10)") "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
          write (u, "(3x,A,ES17.10)") "Q =", object%q
        end if
      end if
      call object%base_write (u)
    else
      write (u, "(1x,A)") "PDF builtin data: [undefined]"
    end if
  end subroutine pdf_builtin_write

```

Initialize. We know that data will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

```

<SF pdf builtin: pdf builtin: TBP>+≡
  procedure :: init => pdf_builtin_init

<SF pdf builtin: procedures>+≡
  subroutine pdf_builtin_init (sf_int, data)
    class(pdf_builtin_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    type(flavor_t) :: flv, flv_remnant
    type(quantum_numbers_t), dimension(3) :: qn
    integer :: i
    select type (data)
    type is (pdf_builtin_data_t)
      mask = new_quantum_numbers_mask (.false., .false., .true.)
      call sf_int%base_init (mask, [0._default], [0._default], [0._default])
      sf_int%data => data
      do i = -6, 6
        if (data%mask(i)) then
          call quantum_numbers_init (qn(1), data%flv_in)
          if (i == 0) then
            call flavor_init (flv, GLUON, data%model)
            call flavor_init (flv_remnant, HADRON_REMNANT_OCTET, data%model)
          else

```

```

        call flavor_init (flv, i, data%model)
        call flavor_init (flv_remnant, &
            sign (HADRON_REMNANT_TRIPLET, -i), data%model)
    end if
    call quantum_numbers_init (qn(2), &
        flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
    call quantum_numbers_init (qn(3), &
        flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
    call interaction_add_state (sf_int%interaction_t, qn)
end if
end do
if (data%has_photon .and. data%mask_photon) then
    call flavor_init (flv, PHOTON, data%model)
    call flavor_init (flv_remnant, HADRON_REMNANT_SINGLET, data%model)
    call quantum_numbers_init (qn(2), flv = flv_remnant, &
        col = color_from_flavor (flv_remnant, 1))
    call quantum_numbers_init (qn(3), flv = flv, &
        col = color_from_flavor (flv, 1, reverse = .true.))
    call interaction_add_state (sf_int%interaction_t, qn)
end if
call interaction_freeze (sf_int%interaction_t)
call sf_int%set_incoming ([1])
call sf_int%set_radiated ([2])
call sf_int%set_outgoing ([3])
sf_int%status = SF_INITIAL
end select
end subroutine pdf_builtin_init

```

#### 10.13.4 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF pdf builtin: pdf builtin: TBP>+=
    procedure :: complete_kinematics => pdf_builtin_complete_kinematics
<SF pdf builtin: procedures>+=
    subroutine pdf_builtin_complete_kinematics (sf_int, x, f, r, map)
        class(pdf_builtin_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(out) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: r
        logical, intent(in) :: map
        real(default) :: xb1
        if (map) then
            call msg_fatal ("PDF builtin: map flag not supported")
        else
            x(1) = r(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        call sf_int%split_momentum (x, xb1)
    end subroutine pdf_builtin_complete_kinematics

```

```

    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end subroutine pdf_builtin_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

⟨SF pdf builtin: pdf builtin: TBP⟩+≡
    procedure :: inverse_kinematics => pdf_builtin_inverse_kinematics

⟨SF pdf builtin: procedures⟩+≡
    subroutine pdf_builtin_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        class(pdf_builtin_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
        real(default) :: xb1
        logical :: set_mom
        set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
        if (map) then
            call msg_fatal ("PDF builtin: map flag not supported")
        else
            r(1) = x(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        if (set_mom) then
            call sf_int%split_momentum (x, xb1)
            select case (sf_int%status)
            case (SF_DONE_KINEMATICS)
                sf_int%x = x(1)
            case (SF_FAILED_KINEMATICS)
                sf_int%x = 0
                f = 0
            end select
        end if
    end subroutine pdf_builtin_inverse_kinematics

```

### 10.13.5 Structure function

Once the scale is also known, we can actually call the PDF and set the values. Contrary to LHAPDF, the wrapper already takes care of adjusting to the  $x$  and  $Q$  bounds. Account for the Jacobian.

```

⟨SF pdf builtin: pdf builtin: TBP⟩+≡
    procedure :: apply => pdf_builtin_apply

```

```

<SF pdf builtin: procedures>+≡
subroutine pdf_builtin_apply (sf_int, scale)
  class(pdf_builtin_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default), dimension(-6:6) :: ff
  real(default) :: x, fph
  complex(default), dimension(:), allocatable :: fc
  associate (data => sf_int%data)
    sf_int%q = scale
    x = sf_int%x
    if (data%invert) then
      if (data%has_photon) then
        call pdf_evolve (data%id, x, scale, ff(6:-6:-1), fph)
      else
        call pdf_evolve (data%id, x, scale, ff(6:-6:-1))
      end if
    else
      if (data%has_photon) then
        call pdf_evolve (data%id, x, scale, ff, fph)
      else
        call pdf_evolve (data%id, x, scale, ff)
      end if
    end if
    if (data%has_photon) then
      allocate (fc (count ([data%mask, data%mask_photon])))
      fc = max (pack ([ff, fph], &
        [data%mask, data%mask_photon]), 0._default)
    else
      allocate (fc (count (data%mask)))
      fc = max (pack (ff, data%mask), 0._default)
    end if
  end associate
  call interaction_set_matrix_element (sf_int%interaction_t, fc)
  sf_int%status = SF_EVALUATED
end subroutine pdf_builtin_apply

```

### 10.13.6 Strong Coupling

Since the PDF codes provide a function for computing the running  $\alpha_s$  value, we make this available as an implementation of the abstract `alpha_qcd_t` type, which is used for matrix element evaluation.

```

<SF pdf builtin: public>+≡
  public :: alpha_qcd_pdf_builtin_t

<SF pdf builtin: types>+≡
  type, extends (alpha_qcd_t) :: alpha_qcd_pdf_builtin_t
    type(string_t) :: pdfset_name
    integer :: pdfset_id = -1
  contains
    <SF pdf builtin: alpha qcd: TBP>
  end type alpha_qcd_pdf_builtin_t

```

Output.

```
<SF pdf builtin: alpha qcd: TBP>≡
  procedure :: write => alpha_qcd_pdf_builtin_write

<SF pdf builtin: procedures>+≡
  subroutine alpha_qcd_pdf_builtin_write (object, unit)
    class(alpha_qcd_pdf_builtin_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A)") "QCD parameters (pdf_builtin):"
    write (u, "(5x,A,A)") "PDF set = ", char (object%pdfset_name)
    write (u, "(5x,A,I0)") "PDF ID = ", object%pdfset_id
  end subroutine alpha_qcd_pdf_builtin_write
```

Calculation: the numeric ID selects the correct PDF set, which must be properly initialized.

```
<SF pdf builtin: alpha qcd: TBP>+≡
  procedure :: get => alpha_qcd_pdf_builtin_get

<SF pdf builtin: procedures>+≡
  function alpha_qcd_pdf_builtin_get (alpha_qcd, scale) result (alpha)
    class(alpha_qcd_pdf_builtin_t), intent(in) :: alpha_qcd
    real(default), intent(in) :: scale
    real(default) :: alpha
    alpha = pdf_alphas (alpha_qcd%pdfset_id, scale)
  end function alpha_qcd_pdf_builtin_get
```

Initialization. We need to access the (quasi-global) initialization status.

```
<SF pdf builtin: alpha qcd: TBP>+≡
  procedure :: init => alpha_qcd_pdf_builtin_init

<SF pdf builtin: procedures>+≡
  subroutine alpha_qcd_pdf_builtin_init (alpha_qcd, status, name, path)
    class(alpha_qcd_pdf_builtin_t), intent(out) :: alpha_qcd
    type(pdf_builtin_status_t), intent(inout) :: status
    type(string_t), intent(in) :: name
    type(string_t), intent(in) :: path
    alpha_qcd%pdfset_name = name
    alpha_qcd%pdfset_id = pdf_get_id (name)
    if (alpha_qcd%pdfset_id < 0) &
      call msg_fatal ("QCD parameter initialization: PDF set " &
        // char (name) // " is unknown")
    call pdf_init (status, alpha_qcd%pdfset_id, path)
  end subroutine alpha_qcd_pdf_builtin_init
```

### 10.13.7 Unit tests

```
<SF pdf builtin: public>+≡
  public :: sf_pdf_builtin_test
```

```

<SF pdf builtin: tests>≡
  subroutine sf_pdf_builtin_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <SF pdf builtin: execute tests>
  end subroutine sf_pdf_builtin_test

```

## Test structure function data

Construct and display a test structure function data object.

```

<SF pdf builtin: execute tests>≡
  call test (sf_pdf_builtin_1, "sf_pdf_builtin_1", &
    "structure function configuration", &
    u, results)

<SF pdf builtin: tests>+≡
  subroutine sf_pdf_builtin_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable :: data
    type(pdf_builtin_status_t) :: status
    type(string_t) :: name

    write (u, "(A)")  "* Test output: sf_pdf_builtin_1"
    write (u, "(A)")  "* Purpose: initialize and display &
      &test structure function data"
    write (u, "(A)")

    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("QCD"), &
      var_str ("QCD.mdl"), os_data, model)
    call flavor_init (flv, PROTON, model)

    allocate (pdf_builtin_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    name = "CTEQ6L"

    select type (data)
    type is (pdf_builtin_data_t)
      call data%init (status, model, flv, name, os_data%pdf_builtin_datapath)
    end select

```

```

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)") "Outgoing particle codes:"
write (u, "(2x,99(1x,I0))") data%get_pdg_out ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_pdf_builtin_1"

end subroutine sf_pdf_builtin_1

```

### Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

```

<SF pdf builtin: execute tests>+≡
call test (sf_pdf_builtin_2, "sf_pdf_builtin_2", &
  "structure function instance", &
  u, results)

<SF pdf builtin: tests>+≡
subroutine sf_pdf_builtin_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(pdf_builtin_status_t) :: status
  type(string_t) :: name
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, x
  real(default) :: f, s

  write (u, "(A)")  "* Test output: sf_pdf_builtin_2"
  write (u, "(A)")  "* Purpose: initialize and fill &
    &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("QCD"), &
    var_str ("QCD.mdl"), os_data, model)
  call flavor_init (flv, PROTON, model)

```

```

call reset_interaction_counter ()

name = "CTEQ6L"

allocate (pdf_builtin_data_t :: data)
select type (data)
type is (pdf_builtin_data_t)
  call data%init (status, model, flv, name, os_data%pdf_builtin_datapath)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (x (size (r)))

r = 0.5_default
call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F12.9))")  "x =", x
write (u, "(A,9(1x,F12.9))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

```



```

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F12.9))") "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate for Q = 100 GeV"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, map=.false.)
call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_pdf_builtin_2"

end subroutine sf_pdf_builtin_2

```

## Strong Coupling

Test  $\alpha_s$  as an implementation of the `alpha_qcd_t` abstract type.

```

<SF pdf builtin: execute tests>+≡
call test (sf_pdf_builtin_3, "sf_pdf_builtin_3", &
  "running alpha_s", &
  u, results)

<SF pdf builtin: tests>+≡
subroutine sf_pdf_builtin_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(pdf_builtin_status_t) :: status
  type(qcd_t) :: qcd
  type(string_t) :: name

  write (u, "(A)")  "* Test output: sf_pdf_builtin_3"
  write (u, "(A)")  "* Purpose: initialize and evaluate alpha_s"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)

  name = "CTEQ6L"

```

```

write (u, "(A)")  "* Initialize qcd object"
write (u, "(A)")

allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
select type (alpha => qcd%alpha)
type is (alpha_qcd_pdf_builtin_t)
  call alpha%init (status, name, os_data%pdf_builtin_datapath)
end select
call qcd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for Q = 100"
write (u, "(A)")

write (u, "(1x,A,F8.5)")  "alpha = ", qcd%alpha%get (100._default)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_pdf_builtin_3"

end subroutine sf_pdf_builtin_3

```

## 10.14 User Plugin for Structure Functions

This variant gives access to user-defined structure functions or spectra.

### 10.14.1 The module

```

⟨sf_user.f90⟩≡
  ⟨File header⟩

module sf_user

  use iso_c_binding !NODEP!
  ⟨Use kinds⟩
  ⟨Use strings⟩
  ⟨Use file utils⟩
  use diagnostics !NODEP!
  use c_particles !NODEP!
  use lorentz !NODEP!
  use subevents
  use user_code_interface
  use models
  use flavors
  use helicities
  use colors
  use quantum_numbers

```

```

    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base

```

*⟨Standard module head⟩*

*⟨SF user: public⟩*

*⟨SF user: types⟩*

contains

*⟨SF user: procedures⟩*

```
end module sf_user
```

## 10.14.2 The user structure function data block

The data block holds the procedure pointers that are used for retrieving static information, as well as the actual evaluation.

*⟨SF user: public⟩*≡

```
public :: user_data_t
```

*⟨SF user: types⟩*≡

```

type, extends(sf_data_t) :: user_data_t
  private
  type(string_t) :: name
  integer :: n_in
  integer :: n_out
  integer :: n_tot
  integer :: n_states
  integer :: n_col
  integer :: n_dim
  integer :: n_var
  integer, dimension(2) :: pdg_in
  type(model_t), pointer :: model => null ()
  procedure(user_int_info), nopass, pointer :: info => null ()
  procedure(user_int_mask), nopass, pointer :: mask => null ()
  procedure(user_int_state), nopass, pointer :: state => null ()
  procedure(user_int_kinematics), nopass, pointer :: kinematics => null ()
  procedure(user_int_evaluate), nopass, pointer :: evaluate => null ()
contains
  ⟨SF user: user data: TBP⟩
end type user_data_t

```

Assign procedure pointers from a dynamically loaded library, given the specified name.

We have to distinguish three cases: (1) Both beams are affected, and the user spectrum implements both beams. There is a single data object. (2) Both

beams are affected, and the user spectrum applies to single beams. Fill two different objects. (3) A single beam is affected.

```

<SF User: public>≡
    public :: sf_user_data_init

<SF User: procedures>≡
    subroutine sf_user_data_init (data, name, flv, model)
        type(sf_user_data_t), intent(out) :: data
        type(string_t), intent(in) :: name
        type(flavor_t), dimension(2), intent(in) :: flv
        type(model_t), intent(in), target :: model
        integer(c_int) :: n_in
        integer(c_int) :: n_out
        integer(c_int) :: n_states
        integer(c_int) :: n_col
        integer(c_int) :: n_dim
        integer(c_int) :: n_var
        data%name = name
        data%pdg_in = flavor_get_pdg (flv)
        data%model => model
        call c_f_procpointer (user_code_find_proc (name // "_info"), data%info)
        call c_f_procpointer (user_code_find_proc (name // "_mask"), data%mask)
        call c_f_procpointer (user_code_find_proc (name // "_state"), data%state)
        call c_f_procpointer &
            (user_code_find_proc (name // "_kinematics"), data%kinematics)
        call c_f_procpointer &
            (user_code_find_proc (name // "_evaluate"), data%evaluate)
        n_in = 1
        n_out = 2
        n_states = 1
        n_col = 2
        n_dim = 1
        n_var = 1
        call data%info (n_in, n_out, n_states, n_col, n_dim, n_var)
        data%n_in = n_in
        data%n_out = n_out
        data%n_tot = n_in + n_out
        data%n_states = n_states
        data%n_col = n_col
        data%n_dim = n_dim
        data%n_var = n_var
    end subroutine sf_user_data_init

```

Output

```

<SF user: user data: TBP>≡
    procedure :: write => user_data_write

<SF user: procedures>≡
    subroutine user_data_write (data, unit, verbose) ! , md5)
        class(user_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        !!! JRR: WK please check: still needed?

```

```

!    logical, intent(in), optional :: md5
    u = output_unit (unit); if (u < 0) return
    write (u, "(1x,A,A)") "User structure function: ", char (data%name)
end subroutine user_data_write

```

Retrieving contents

*<SF User: public>+≡*

```

public :: sf_user_data_get_name

```

*<SF User: procedures>+≡*

```

function sf_user_data_get_name (data) result (name)
    type(string_t) :: name
    type(sf_user_data_t), intent(in) :: data
    name = data%name
end function sf_user_data_get_name

```

*<SF User: public>+≡*

```

public :: sf_user_data_get_n_in
public :: sf_user_data_get_n_out
public :: sf_user_data_get_n_tot
public :: sf_user_data_get_n_dim
public :: sf_user_data_get_n_var

```

*<SF User: procedures>+≡*

```

function sf_user_data_get_n_in (data) result (n_in)
    integer :: n_in
    type(sf_user_data_t), intent(in) :: data
    n_in = data%n_in
end function sf_user_data_get_n_in

```

```

function sf_user_data_get_n_out (data) result (n_out)
    integer :: n_out
    type(sf_user_data_t), intent(in) :: data
    n_out = data%n_out
end function sf_user_data_get_n_out

```

```

function sf_user_data_get_n_tot (data) result (n_tot)
    integer :: n_tot
    type(sf_user_data_t), intent(in) :: data
    n_tot = data%n_tot
end function sf_user_data_get_n_tot

```

```

function sf_user_data_get_n_dim (data) result (n_dim)
    integer :: n_dim
    type(sf_user_data_t), intent(in) :: data
    n_dim = data%n_dim
end function sf_user_data_get_n_dim

```

```

function sf_user_data_get_n_var (data) result (n_var)
    integer :: n_var
    type(sf_user_data_t), intent(in) :: data
    n_var = data%n_var
end function sf_user_data_get_n_var

```

### 10.14.3 The interaction

We fill the interaction by looking up the table of states using the interface functions.

For particles which have a known flavor (as indicated by the mask), we compute the mass squared, so we can use it for the invariant mass of the particle objects.

```
<SF user: user: TBP>≡
  procedure :: init => user_init

<SF user: procedures>+≡
  subroutine user_init (sf_int, data)
    !!! JRR: WK please check
    class(user_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
    integer, dimension(:), allocatable :: hel_lock
    integer(c_int) :: m_flv, m_hel, m_col, i_lock
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer(c_int) :: f, h
    integer(c_int), dimension(:), allocatable :: c
    type(flavor_t) :: flv
    type(helicity_t) :: hel
    type(color_t) :: col
    integer :: i, s
    integer(c_int) :: i_prt, i_state
    select type (data)
    type is (user_data_t)
      allocate (mask (data%n_tot))
      allocate (hel_lock (data%n_tot))
      allocate (qn (data%n_tot))
      allocate (c (data%n_col))
      do i = 1, size (mask)
        i_prt = i
        m_flv = 0; m_col = 0; m_hel = 0; i_lock = 0
        call data%mask (i_prt, m_flv, m_col, m_hel, i_lock)
        mask(i) = &
          new_quantum_numbers_mask (m_flv /= 0, m_col /= 0, m_hel /= 0)
        hel_lock(i) = i_lock
      end do
      !!! Will have to be filled in later.
      ! call sf_int%base_init (mask, &
      !   hel_lock = hel_lock)
      call interaction_init &
        (sf_int%interaction_t, data%n_in, 0, data%n_out, mask=mask, &
        hel_lock=hel_lock, set_relations=.true.)
      do s = 1, data%n_states
        i_state = s
        do i = 1, data%n_tot
          i_prt = i
          f = 0; h = 0; c = 0
          call data%state (i_state, i_prt, f, h, c)
          if (m_flv == 0) then
            call flavor_init (flv, int (f), data%model)
```

```

        else
            call flavor_init (flv)
        end if
        if (m_hel == 0) then
            call helicity_init (hel, int (h))
        else
            call helicity_init (hel)
        end if
        if (m_col == 0) then
            call color_init_from_array (col, int (c))
        else
            call color_init (col)
        end if
        call quantum_numbers_init (qn(i), flv, col, hel)
    end do
    call interaction_add_state (sf_int%interaction_t, qn)
end do
call interaction_freeze (sf_int%interaction_t)
!!! JRR: WK please check
!!! What has to be inserted here?
! call sf_int%set_incoming (??)
! call sf_int%set_radiated (??)
! call sf_int%set_outgoing (??)
sf_int%status = SF_INITIAL
end select
end subroutine user_init

```

Allocate the interaction record.

```

<SF user: user data: TBP>+≡
    procedure :: allocate_sf_int => user_data_allocate_sf_int

<SF user: procedures>+≡
    subroutine user_data_allocate_sf_int (data, sf_int)
        class(user_data_t), intent(in) :: data
        class(sf_int_t), intent(inout), allocatable :: sf_int
        allocate (user_t :: sf_int)
    end subroutine user_data_allocate_sf_int

```

The number of parameters is one. We do not generate transverse momentum.

```

<SF user: user data: TBP>+≡
    procedure :: get_n_par => user_data_get_n_par

<SF user: procedures>+≡
    function user_data_get_n_par (data) result (n)
        class(user_data_t), intent(in) :: data
        integer :: n
        n = data%n_var
    end function user_data_get_n_par

```

Return the outgoing particle PDG codes. This has to be inferred from the states (right?). JRR: WK please check.

```

<SF user: user data: TBP>+≡
    procedure :: get_pdg_out => user_data_get_pdg_out

```

```

<SF user: procedures>+≡
function user_data_get_pdg_out (data) result (pdg_out)
  class(user_data_t), intent(in) :: data
  integer, dimension(:), allocatable :: pdg_out
  !!! integer :: n, np, i
  !!! n = count (data%mask)
  !!! np = 0; if (data%has_photon .and. data%mask_photon) np = 1
  !!! allocate (pdg_out (n + np))
  !!! pdg_out(1:n) = pack ([i, i = -6, 6]), data%mask)
  !!! if (np == 1) pdg_out(n+np) = PHOTON
end function user_data_get_pdg_out

```

#### 10.14.4 The user structure function

For maximal flexibility, user structure functions separate kinematics from dynamics just as the PDF interface does. (JRR: Ok, I guess this now done for all structure functions, right?) We create `c_prt_t` particle objects from the incoming momenta (all other quantum numbers are irrelevant) and call the user-supplied kinematics function to compute the outgoing momenta, along with other variables that will be needed for matrix element evaluation. If known, we use the mass squared computed above. !!! JRR: WK please check I don't know actually whether this really fits into the setup done by WK.

```

<SF user: types>+≡
!!! JRR: WK please check
type, extends (sf_int_t) :: user_t
  type(user_data_t), pointer :: data => null ()
  real(default) :: x = 0
  real(default) :: q = 0
contains
  <SF user: user: TBP>
end type user_t

```

Type string: display the name of the user structure function.

```

<SF user: user: TBP>+≡
procedure :: type_string => user_type_string

<SF user: procedures>+≡
function user_type_string (object) result (string)
  class(user_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
    string = "User structure function: " // object%data%name
  else
    string = "User structure function: [undefined]"
  end if
end function user_type_string

```

Output. Call the interaction routine after displaying the configuration.

```

<SF user: user: TBP>+≡
procedure :: write => user_write

```



```

<SF user: procedures>+≡
subroutine user_write (object, unit)
  !!! JRR: WK please check
  !!! Guess these variables do not exist for CIRCE1 (?)
  class(user_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
    call object%data%write (u)
    if (object%status >= SF_DONE_KINEMATICS) then
      write (u, "(1x,A)") "SF parameters:"
      write (u, "(3x,A,ES17.10)") "x =", object%x
      if (object%status >= SF_FAILED_EVALUATION) then
        write (u, "(3x,A,ES17.10)") "Q =", object%q
      end if
    end if
    call object%base_write (u)
  else
    write (u, "(1x,A)") "User structure function data: [undefined]"
  end if
end subroutine user_write

```

#### 10.14.5 Kinematics

Set kinematics. If `map` is unset, the  $r$  and  $x$  values coincide, and the Jacobian  $f(r)$  is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set  $x = r^2$  and consequently  $f(r) = 2r$ .

```

<SF user: user: TBP>+≡
  procedure :: complete_kinematics => user_complete_kinematics

<SF user: procedures>+≡
subroutine user_complete_kinematics (sf_int, x, f, r, map)
  !!! JRR: WK please check
  !!! This cannot be correct, as the CIRCE1 structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(user_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
    call msg_fatal ("User structure function: map flag not supported")
  else
    x(1) = r(1)
    f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)

```

```

        sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
    end select
end subroutine user_complete_kinematics

```

Compute inverse kinematics. Here, we start with the  $x$  array and compute the “input”  $r$  values and the Jacobian  $f$ . After this, we can set momenta by the same formula as for normal kinematics.

```

<SF user: user: TBP>+≡
    procedure :: inverse_kinematics => user_inverse_kinematics

<SF user: procedures>+≡
    subroutine user_inverse_kinematics (sf_int, x, f, r, map, set_momenta)
        !!! JRR: WK please check
        !!! This cannot be correct, as the CIRCE1 structure function has
        !!! twice the variables (2->4 instead of 1->2 splitting)
        class(user_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: r
        logical, intent(in) :: map
        logical, intent(in), optional :: set_momenta
        real(default) :: xb1
        logical :: set_mom
        set_mom = .false.; if (present (set_momenta)) set_mom = set_momenta
        if (map) then
            call msg_fatal ("User structure function: map flag not supported")
        else
            r(1) = x(1)
            f = 1
        end if
        xb1 = 1 - x(1)
        if (set_mom) then
            call sf_int%split_momentum (x, xb1)
            select case (sf_int%status)
            case (SF_DONE_KINEMATICS)
                sf_int%x = x(1)
            case (SF_FAILED_KINEMATICS)
                sf_int%x = 0
                f = 0
            end select
        end if
    end subroutine user_inverse_kinematics

<SF User: public>+≡
    public :: interaction_set_kinematics_sf_user

<SF User: procedures>+≡
    subroutine interaction_set_kinematics_sf_user (int, x, r, data)
        type(interaction_t), intent(inout) :: int
        real(default), dimension(:), intent(out) :: x
        real(default), dimension(:), intent(in) :: r
    end subroutine interaction_set_kinematics_sf_user

```

```

type(sf_user_data_t), intent(in) :: data
type(vector4_t), dimension(data%n_in) :: p_in
type(vector4_t), dimension(data%n_out) :: p_out
type(c_prt_t), dimension(data%n_in) :: prt_in
type(c_prt_t), dimension(data%n_out) :: prt_out
real(c_double), dimension(data%n_var) :: xval
call interaction_get_momenta_sub (int, p_in, outgoing=.false.)
prt_in = vector4_to_c_prt (p_in)
prt_in%type = PRT_INCOMING
call data%kinematics (prt_in, real (r, c_double), prt_out, xval)
x = xval
p_out = vector4_from_c_prt (prt_out)
call interaction_set_momenta (int, p_out, outgoing=.true.)
end subroutine interaction_set_kinematics_sf_user

```

The matrix-element evaluation may require a scale parameter, therefore this routine is separate. We take the variables computed above together with the event energy scale and call the user function that computes the matrix elements.

```

<SF user: user: TBP>+≡
  procedure :: apply => user_apply

<SF user: procedures>+≡
  subroutine user_apply (sf_int, scale) !, x, data)
    !!! JRR: WK please check
    class(user_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default), dimension(:), allocatable :: x
    real(c_double), dimension(sf_int%data%n_states) :: fval
    complex(default), dimension(sf_int%data%n_states) :: fc
    associate (data => sf_int%data)
      !!! This is wrong, has to be replaced
      ! allocate (x, size (sf_int%x))
      x = sf_int%x
      call data%evaluate (real (x, c_double), real (scale, c_double), fval)
      fc = fval
      call interaction_set_matrix_element (sf_int%interaction_t, fc)
    end associate
    sf_int%status = SF_EVALUATED
  end subroutine user_apply

```

## 10.15 Spectra and structure functions: wrapper

In this module, we collect, for each type of spectrum or structure function, the data, initialization routines, and applications.

```

<strfun.f90>≡
  <File header>

  module strfun

    <Use kinds>
    <Use strings>

```

```

<Use file utils>
!! !! use diagnostics !NODEP!
!! !! use lorentz !NODEP!
!! !! use models
!! !! use quantum_numbers
!! !! use interactions
!! !! use evaluators
!! !! use beams
!! !! use sf_isr
!! !! use sf_epa
!! !! use sf_ewa
!! !! use sf_circe1
!! !! use sf_circe2
!! !! use sf_escan
!! !! use sf_beam_events
!! !! use sf_lhapdf
!! !! use sf_pdf_builtin
!! !! use sf_user

```

```

<Standard module head>

```

```

<Strfun: public>

```

```

<Strfun: parameters>

```

```

<Strfun: types>

```

```

<Strfun: interfaces>

```

```

contains

```

```

<Strfun: procedures>

```

```

end module strfun

```

## 10.15.1 The structure functions type

### Definition

This contains the specific structure function data, much of which depends on the type. An extensible type would be appropriate. As long as this is not available in general, we emulate it by allocating the requested data explicitly.

```

<Strfun: types>≡
!! !! type :: strfun_t
!! !!     private
!! !!     integer :: type = STRF_NONE
!! !!     type(string_t) :: name
!! !!     type(interaction_t) :: int
!! !!     type(lhapdf_data_t), dimension(:), allocatable :: lhpdf_data
!! !!     type(pdf_builtin_data_t), dimension(:), allocatable :: pdf_builtin_data
!! !!     type(isr_data_t), dimension(:), allocatable :: isr_data
!! !!     type(epa_data_t), dimension(:), allocatable :: epa_data
!! !!     type(ewa_data_t), dimension(:), allocatable :: ewa_data
!! !!     type(circe1_data_t), dimension(:), allocatable :: circe1_data

```

```

!! !!   type(circe2_data_t), dimension(:), allocatable :: circe2_data
!! !!   type(escan_data_t), dimension(:), allocatable :: escan_data
!! !!   type(beam_events_data_t), dimension(:), allocatable :: beam_events_data
!! !!   type(sf_user_data_t), dimension(:), allocatable :: user_data
!! !!   real(default) :: x = 0, f = 1, s = 0
!! !!   real(default), dimension(:), allocatable :: user_xval
!! !!   real(default) :: scale = 0
!! !! end type strfun_t

```

The list of structure function codes:

```

<Strfun: parameters>≡
integer, parameter, public :: STRF_NONE = 0
integer, parameter, public :: STRF_LHAPDF = 1, STRF_ISR = 2, &
    STRF_EPA = 3, STRF_EWA = 4, STRF_CIRCE1 = 5, STRF_CIRCE2 = 6, &
    STRF_ESCAN = 7, STRF_BEVT = 8, STRF_PDF_BUILTIN = 9
integer, parameter, public :: STRF_USER = 99

```

The initializer assigns specific data and tags the interaction. The data block(s) have to be known already.

```

<Strfun: interfaces>≡
interface strfun_init
    module procedure strfun_init_lhapdf
    module procedure strfun_init_isr
    module procedure strfun_init_epa
    module procedure strfun_init_ewa
    module procedure strfun_init_circe1
    module procedure strfun_init_circe2
    module procedure strfun_init_escan
    module procedure strfun_init_beam_events
    module procedure strfun_init_pdf_builtin
    module procedure strfun_init_user
end interface

```

```

<Strfun: procedures>≡
!! !!   subroutine strfun_init_lhapdf (strfun, lhpdf_data)
!! !!       type(strfun_t), intent(out) :: strfun
!! !!       type(lhpdf_data_t), intent(in) :: lhpdf_data
!! !!       strfun%type = STRF_LHAPDF
!! !!       strfun%name = "LHAPDF"
!! !!       allocate (strfun%lhpdf_data (1))
!! !!       strfun%lhpdf_data = lhpdf_data
!! !!       call interaction_init_lhapdf (strfun%int, lhpdf_data)
!! !!   end subroutine strfun_init_lhapdf
!! !!
!! !!   subroutine strfun_init_pdf_builtin (strfun, pdf_builtin_data)
!! !!       type(strfun_t), intent(out) :: strfun
!! !!       type(pdf_builtin_data_t), intent(in) :: pdf_builtin_data
!! !!       strfun%type = STRF_PDF_BUILTIN
!! !!       strfun%name = "builtin PDF: " // pdf_builtin_get_name (pdf_builtin_data)
!! !!       allocate (strfun%pdf_builtin_data (1))
!! !!       strfun%pdf_builtin_data = pdf_builtin_data
!! !!       call interaction_init_pdf_builtin (strfun%int, pdf_builtin_data)

```

```

!! !! end subroutine strfun_init_pdf_builtin
!! !!
!! !! subroutine strfun_init_isr (strfun, isr_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(isr_data_t), intent(in) :: isr_data
!! !!   strfun%type = STRF_ISR
!! !!   strfun%name = "ISR"
!! !!   allocate (strfun%isr_data (1))
!! !!   strfun%isr_data = isr_data
!! !!   call interaction_init_isr (strfun%int, isr_data)
!! !! end subroutine strfun_init_isr
!! !!
!! !! subroutine strfun_init_epa (strfun, epa_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(epa_data_t), intent(in) :: epa_data
!! !!   strfun%type = STRF_EPA
!! !!   strfun%name = "EPA"
!! !!   allocate (strfun%epa_data (1))
!! !!   strfun%epa_data = epa_data
!! !!   call interaction_init_epa (strfun%int, epa_data)
!! !! end subroutine strfun_init_epa
!! !!
!! !! subroutine strfun_init_ewa (strfun, ewa_data, id)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(ewa_data_t), intent(inout) :: ewa_data
!! !!   integer, intent(in) :: id
!! !!   strfun%type = STRF_EWA
!! !!   strfun%name = "EWA"
!! !!   allocate (strfun%ewa_data (1))
!! !!   call ewa_set_id (ewa_data, id)
!! !!   strfun%ewa_data = ewa_data
!! !!   call interaction_init_ewa (strfun%int, ewa_data)
!! !! end subroutine strfun_init_ewa
!! !!
!! !! subroutine strfun_init_circe1 (strfun, circe1_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(circe1_data_t), intent(in) :: circe1_data
!! !!   strfun%type = STRF_CIRCE1
!! !!   strfun%name = "CIRCE1"
!! !!   allocate (strfun%circe1_data (1))
!! !!   strfun%circe1_data = circe1_data
!! !!   call interaction_init_circe1 (strfun%int, circe1_data)
!! !! end subroutine strfun_init_circe1
!! !!
!! !! subroutine strfun_init_circe2 (strfun, circe2_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(circe2_data_t), intent(in) :: circe2_data
!! !!   strfun%type = STRF_CIRCE2
!! !!   strfun%name = "CIRCE2"
!! !!   allocate (strfun%circe2_data (1))
!! !!   strfun%circe2_data = circe2_data
!! !!   call interaction_init_circe2 (strfun%int, circe2_data)
!! !! end subroutine strfun_init_circe2
!! !!

```

```

!! !! subroutine strfun_init_escan (strfun, escan_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(escan_data_t), intent(in) :: escan_data
!! !!   strfun%type = STRF_ESCAN
!! !!   strfun%name = "Energy scan"
!! !!   allocate (strfun%escan_data (1))
!! !!   strfun%escan_data = escan_data
!! !!   call interaction_init_escan (strfun%int, escan_data)
!! !! end subroutine strfun_init_escan
!! !!
!! !! subroutine strfun_init_beam_events (strfun, beam_events_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(beam_events_data_t), intent(in) :: beam_events_data
!! !!   strfun%type = STRF_BEVT
!! !!   strfun%name = "Beam events"
!! !!   allocate (strfun%beam_events_data (1))
!! !!   strfun%beam_events_data = beam_events_data
!! !!   call interaction_init_beam_events (strfun%int, beam_events_data)
!! !! end subroutine strfun_init_beam_events
!! !!
!! !! subroutine strfun_init_user (strfun, user_data)
!! !!   type(strfun_t), intent(out) :: strfun
!! !!   type(sf_user_data_t), intent(in) :: user_data
!! !!   strfun%type = STRF_USER
!! !!   strfun%name = "User structure function: " &
!! !!     // sf_user_data_get_name (user_data)
!! !!   allocate (strfun%user_data (1))
!! !!   strfun%user_data = user_data
!! !!   call interaction_init_sf_user (strfun%int, user_data)
!! !!   allocate (strfun%user_xval (sf_user_data_get_n_var (user_data)))
!! !! end subroutine strfun_init_user
!! !!

```

Finalizer for the contained interaction. The presence of file operations forbid the `elemental` attribute, therefore the interface.

```

<Strfun: interfaces>+≡
  interface strfun_final
    module procedure strfun_final0
    module procedure strfun_final1
  end interface

<Strfun: procedures>+≡
  subroutine strfun_final1 (strfun)
    type(strfun_t), dimension(:), intent(inout) :: strfun
    integer :: i
    do i = 1, size (strfun)
      call strfun_final0 (strfun(i))
    end do
  end subroutine strfun_final1

  subroutine strfun_final0 (strfun)
    type(strfun_t), intent(inout) :: strfun
    select case (strfun%type)
    case (STRF_ISR)
      deallocate (strfun%isr_data)

```

```

case (STRF_EPA)
  deallocate (strfun%epa_data)
case (STRF_EWA)
  deallocate (strfun%ewa_data)
case (STRF_CIRCE1)
  deallocate (strfun%circe1_data)
case (STRF_CIRCE2)
  deallocate (strfun%circe2_data)
case (STRF_ESCAN)
  deallocate (strfun%escan_data)
case (STRF_BEVT)
  call beam_events_data_close (strfun%beam_events_data(1))
  deallocate (strfun%beam_events_data)
case (STRF_LHAPDF)
  deallocate (strfun%lhpdf_data)
case (STRF_PDF_BUILTIN)
  call pdf_builtin_final (strfun%pdf_builtin_data(1))
  deallocate (strfun%pdf_builtin_data)
case (STRF_USER)
  deallocate (strfun%user_data)
  deallocate (strfun%user_xval)
end select
call interaction_final (strfun%int)
strfun%type = STRF_NONE
end subroutine strfun_final0

```

## I/O

*(Strfun: procedures)*+≡

```

!!subroutine strfun_write (strfun, unit, verbose, show_momentum_sum, show_mass)
!!  type(strfun_t), intent(in) :: strfun
!!  integer, intent(in), optional :: unit
!!  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
!!  integer :: u
!!  u = output_unit (unit); if (u < 0) return
!!  if (strfun%type /= STRF_NONE) then
!!    write (u, *) char (strfun_get_name (strfun)) // " setup:"
!!    select case (strfun%type)
!!      case (STRF_LHAPDF)
!!        call lhpdf_data_write (strfun%lhpdf_data(1), u)
!!        write (u, *) "LHAPDF event data:"
!!        write (u, *) " x      =", strfun%x
!!        write (u, *) " f      =", strfun%f
!!        write (u, *) " scale =", strfun%scale
!!        write (u, *) " p2     =", strfun%s
!!      case (STRF_PDF_BUILTIN)
!!        call pdf_builtin_data_write (strfun%pdf_builtin_data(1), u)
!!        write (u, *) "PDF event data:"
!!        write (u, *) " x      =", strfun%x
!!        write (u, *) " f      =", strfun%f
!!        write (u, *) " scale =", strfun%scale
!!        write (u, *) " p2     =", strfun%s
!!    case (STRF_ISR)

```



```

!!      call isr_data_write (strfun%isr_data(1), u)
!!    case (STRF_EPA)
!!      call epa_data_write (strfun%epa_data(1), u)
!!    case (STRF_EWA)
!!      call ewa_data_write (strfun%ewa_data(1), u)
!!    case (STRF_CIRCE1)
!!      call circe1_data_write (strfun%circe1_data(1), u)
!!    case (STRF_CIRCE2)
!!      call circe2_data_write (strfun%circe2_data(1), u)
!!    case (STRF_ESCAN)
!!      call escan_data_write (strfun%escan_data(1), u)
!!    case (STRF_BEVT)
!!      call beam_events_data_write (strfun%beam_events_data(1), u)
!!    case (STRF_USER)
!!      call sf_user_data_write (strfun%user_data(1), u)
!!      write (u, *) "User event data:"
!!      if (allocated (strfun%user_xval)) then
!!        write (u, *) "  x      =", strfun%user_xval
!!      else
!!        write (u, *) "  x      = [not allocated]"
!!      end if
!!      write (u, *) "  scale =", strfun%scale
!!    end select
!!    call interaction_write &
!!      (strfun%int, unit, verbose, show_momentum_sum, show_mass)
!!  else
!!    write (u, *) "Structure function setup: [empty]"
!!  end if
!!end subroutine strfun_write

```

## Retrieve data

*<Strfun: procedures>+≡*

```

function strfun_get_name (strfun) result (name)
  type(string_t) :: name
  type(strfun_t), intent(in) :: strfun
  name = strfun%name
end function strfun_get_name

```

*<Strfun: procedures>+≡*

```

function strfun_get_type (strfun) result (type)
  integer :: type
  type(strfun_t), intent(in) :: strfun
  type = strfun%type
end function strfun_get_type

```

## Apply structure function

Set kinematics using input random numbers. For some structure functions, we can already compute matrix elements.

The `no_map` flag implies that all preset mappings and generator modes should be switched off, and the structure functions should be probed directly.

*(Strfun: procedures)* +=

```
!! !! subroutine strfun_set_kinematics (strfun, r, no_map)
!! !!   type(strfun_t), intent(inout) :: strfun
!! !!   real(default), dimension(:), intent(in) :: r
!! !!   logical, intent(in) :: no_map
!! !!   select case (strfun%type)
!! !!     case (STRF_LHAPDF)
!! !!       call interaction_set_kinematics_lhapdf (strfun%int, &
!! !!         strfun%x, strfun%f, strfun%s, r(1), strfun%lhpdf_data(1))
!! !!     case (STRF_PDF_BUILTIN)
!! !!       call interaction_set_kinematics_pdf_builtin (strfun%int, &
!! !!         strfun%x, strfun%f, strfun%s, r(1), strfun%pdf_builtin_data(1))
!! !!     case (STRF_ISR)
!! !!       call interaction_apply_isr (strfun%int, r, strfun%isr_data(1), no_map)
!! !!     case (STRF_EPA)
!! !!       call interaction_apply_epa (strfun%int, r, strfun%epa_data, no_map)
!! !!     case (STRF_EWA)
!! !!       call interaction_apply_ewa (strfun%int, r, strfun%ewa_data, no_map)
!! !!     case (STRF_CIRCE1)
!! !!       call interaction_apply_circe1 &
!! !!         (strfun%int, r, strfun%circe1_data(1), no_map)
!! !!     case (STRF_CIRCE2)
!! !!       call interaction_apply_circe2 &
!! !!         (strfun%int, r, strfun%circe2_data(1), no_map)
!! !!     case (STRF_ESCAN)
!! !!       call interaction_apply_escan &
!! !!         (strfun%int, r, strfun%escan_data(1))
!! !!     case (STRF_BEVT)
!! !!       call interaction_apply_beam_events &
!! !!         (strfun%int, strfun%beam_events_data(1))
!! !!     case (STRF_USER)
!! !!       call interaction_set_kinematics_sf_user (strfun%int, &
!! !!         strfun%user_xval, r, strfun%user_data(1))
!! !!   end select
!! !! end subroutine strfun_set_kinematics
```

Set values where they depend on a separate energy scale:

*(Strfun: procedures)* +=

```
subroutine strfun_apply (strfun, scale)
  type(strfun_t), intent(inout) :: strfun
  real(default), intent(in) :: scale
  strfun%scale = scale
  select case (strfun%type)
    !! !! case (STRF_LHAPDF)
    !! !!   call interaction_apply_lhapdf (strfun%int, scale, &
    !! !!     strfun%x, strfun%f, strfun%s, strfun%lhpdf_data(1))
    !! !! case (STRF_PDF_BUILTIN)
    !! !!   call interaction_apply_pdf_builtin (strfun%int, scale, &
    !! !!     strfun%x, strfun%f, strfun%s, strfun%pdf_builtin_data(1))
    case (STRF_USER)
      call interaction_apply_sf_user (strfun%int, scale, &
```

```

        strfun%user_xval, strfun%user_data(1))
    end select
end subroutine strfun_apply

```

## 10.15.2 Mappings

### Definition

Mappings for single structure functions may be defined in the individual sections, but pairwise mappings belong here. We define a mapping type that applies to an array of  $x$  parameters identified by their indices. The individual mapping types are identified by a **type** parameter. A mapping may depend on a set on real parameters.

```

<Strfun: parameters>+≡
    integer, parameter, public :: SFM_NONE = 0
    integer, parameter, public :: SFM_PAIR = 1
    integer, parameter, public :: SFM_PAIR_RESONANCE = 2

<Strfun: types>+≡
    type :: strfun_mapping_t
    private
    integer, dimension(:), allocatable :: index
    integer :: type = SFM_NONE
    real(default) :: p = 0
    real(default) :: m2 = 0, mg = 0, s = 0
    real(default) :: a1 = 0, a2 = 0, a3 = 0
end type strfun_mapping_t

```

Initialization:

```

<Strfun: procedures>+≡
    subroutine strfun_mapping_init (sf_mapping, index, type, par)
    type(strfun_mapping_t), intent(out) :: sf_mapping
    integer, dimension(:), intent(in) :: index
    integer, intent(in) :: type
    real(default), dimension(:), intent(in) :: par
    real(default) :: s, m2, mg
    allocate (sf_mapping%index (size (index)))
    sf_mapping%index = index
    sf_mapping%type = type
    select case (type)
    case (SFM_PAIR)
        sf_mapping%p = par(1)
    case (SFM_PAIR_RESONANCE)
        s = par(1)**2
        m2 = par(2)**2
        mg = par(2) * par(3)
        sf_mapping%s = s
        sf_mapping%m2 = m2
        sf_mapping%mg = mg
        sf_mapping%a1 = atan (- m2 / mg)
        sf_mapping%a2 = atan ((s - m2) / mg)
        sf_mapping%a3 = (sf_mapping%a2 - sf_mapping%a1) * mg / s
    end select
end subroutine strfun_mapping_init

```

```

end select
end subroutine strfun_mapping_init

```

Output

*(Strfun: procedures)*+≡

```

subroutine strfun_mapping_write (sf_mapping, unit)
  type(strfun_mapping_t), intent(in) :: sf_mapping
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  write (u, "(1x,A)", advance="no") "Strfun mapping for indices: "
  write (u, "(10(1x,I0))") sf_mapping%index
  write (u, "(1x,A,1x,I0)") "mapping type =", sf_mapping%type
  write (u, "(1x,A)", advance="no") "mapping pars ="
  select case (sf_mapping%type)
  case (SFM_NONE)
    write (u, *) "[none]"
  case (SFM_PAIR)
    write (u, *) sf_mapping%p
  case (SFM_PAIR_RESONANCE)
    write (u, *) sf_mapping%s, sf_mapping%m2, sf_mapping%mg
  end select
end subroutine strfun_mapping_write

```

Evaluation

*(Strfun: procedures)*+≡

```

subroutine strfun_mapping_apply (sf_mapping, x, factor)
  type(strfun_mapping_t), intent(in) :: sf_mapping
  real(default), dimension(:), intent(inout) :: x
  real(default), intent(out) :: factor
  real(default) :: f1, f2
  real(default), dimension(2) :: x2
  select case (sf_mapping%type)
  case (SFM_PAIR)
    x2 = x(sf_mapping%index)
    call map_unit_square (x2, factor, sf_mapping%p)
    x(sf_mapping%index) = x2
  case (SFM_PAIR_RESONANCE)
    x2 = x(sf_mapping%index)
    call map_resonance (x2(1), f2, &
      sf_mapping%s, sf_mapping%m2, sf_mapping%mg, &
      sf_mapping%a1, sf_mapping%a2, sf_mapping%a3)
    call map_unit_square (x2, f1)
    x(sf_mapping%index) = x2
    factor = f1 * f2
  case default
    factor = 1
  end select
end subroutine strfun_mapping_apply

subroutine strfun_mapping_apply_inverse (sf_mapping, x, factor)

```

```

type(strfun_mapping_t), intent(in) :: sf_mapping
real(default), dimension(:), intent(inout) :: x
real(default), intent(out) :: factor
real(default) :: f1, f2
real(default), dimension(2) :: x2
select case (sf_mapping%type)
case (SFM_PAIR)
  x2 = x(sf_mapping%index)
  call map_unit_square_inverse (x2, factor, sf_mapping%p)
  x(sf_mapping%index) = x2
case (SFM_PAIR_RESONANCE)
  x2 = x(sf_mapping%index)
  call map_unit_square_inverse (x2, f1)
  call map_resonance_inverse (x2(1), f2, &
    sf_mapping%s, sf_mapping%m2, sf_mapping%mg, &
    sf_mapping%a1, sf_mapping%a2, sf_mapping%a3)
  x(sf_mapping%index) = x2
  factor = f1 * f2
case default
  factor = 1
end select
end subroutine strfun_mapping_apply_inverse

```

This mapping of the unit square is appropriate in particular for structure functions which are concentrated at the lower end. Instead of a rectangular grid, one set of grid lines corresponds to constant parton c.m. energy. The other set is chosen such that the jacobian is only mildly singular ( $\ln x$  which is zero at  $x = 1$ ), corresponding to an initial concentration of sampling points at the maximum energy. If **power** is greater than one (the default), points are also concentrated at the lower end.

*(Strfun: procedures)*+≡

```

subroutine map_unit_square (x, factor, power)
  real(kind=default), dimension(2), intent(inout) :: x
  real(kind=default), intent(out) :: factor
  real(kind=default), intent(in), optional :: power
  real(kind=default) :: xx, yy
  factor = 1
  xx = x(1)
  yy = x(2)
  if (present(power)) then
    if (x(1) > 0 .and. power > 1) then
      xx = x(1)**power
      factor = factor * power * xx / x(1)
    end if
  end if
  if (xx /= 0) then
    x(1) = xx ** yy
    x(2) = xx / x(1)
    factor = factor * abs (log (xx))
  else
    x = 0
  end if
end subroutine map_unit_square

```

This is the inverse mapping.

*<Strfun: procedures>+≡*

```

subroutine map_unit_square_inverse (x, factor, power)
  real(kind=default), dimension(2), intent(inout) :: x
  real(kind=default), intent(out) :: factor
  real(kind=default), intent(in), optional :: power
  real(kind=default) :: lg, xx, yy
  factor = 1
  xx = x(1) * x(2)
  if (xx /= 0) then
    lg = log (xx)
    yy = log (x(1)) / lg
    x(2) = yy
    factor = factor * abs (lg)
    if (present(power)) then
      x(1) = xx**(1._default/power)
      factor = factor * power * xx / x(1)
    else
      x(1) = xx
    end if
  else
    x = 0
  end if
end subroutine map_unit_square_inverse

```

This is the usual mapping of a Lorentz peak. Since the  $x$  range maps the energy range  $0 \dots \sqrt{s}$  to  $0 \dots 1$ , the position of the peak is  $m^2/s$ , the width is given by  $m\Gamma/s$ .

*<Strfun: procedures>+≡*

```

subroutine map_resonance (x, factor, s, m2, mg, a1, a2, a3)
  real(default), intent(inout) :: x
  real(default), intent(out) :: factor
  real(default), intent(in) :: s, m2, mg, a1, a2, a3
  real(default) :: t, z
  z = (1 - x) * a1 + x * a2
  t = tan (z)
  x = (m2 + t * mg) / s
  factor = a3 * (1 + t**2)
end subroutine map_resonance

```

*<Strfun: procedures>+≡*

```

subroutine map_resonance_inverse (x, factor, s, m2, mg, a1, a2, a3)
  real(default), intent(inout) :: x
  real(default), intent(out) :: factor
  real(default), intent(in) :: s, m2, mg, a1, a2, a3
  real(default) :: t
  t = (x * s - m2) / mg
  x = (atan (t) - a1) / (a2 - a1)
  factor = a3 * (1 + t**2)
end subroutine map_resonance_inverse

```

### 10.15.3 Structure function chains

#### Definition

The structure function chain contains an array of structure functions, where each one has one or more free parameters. For each structure function there is an interaction, an image of the interaction within the `strfun` object, which is needed when quantum numbers are reduced. Furthermore, an array of evaluators which cumulatively multiply the structure functions. The last evaluator is connected to the hard matrix element.

The `last_strfun` and `out_index` index (pairs) identify, for each beam, the last structure function and the outgoing particle. The `coll_index` (pair) identifies the outgoing particles in the last evaluator.

For a decay, this structure is also used, but normally there are no structure functions besides the “beam” object.

```

(Strfun: public)≡
    public :: strfun_chain_t

(Strfun: types)+≡
    type :: strfun_chain_t
        private
        type(beam_t) :: beam
        integer :: n_strfun = 0
        logical :: multichannel = .false.
        integer :: n_mapping = 0
        type(strfun_t), dimension(:), allocatable :: strfun
        type(strfun_mapping_t), dimension(:, :), allocatable :: sf_mapping
        real(default) :: mapping_factor = 0
        integer :: n_parameters_tot = 0
        integer, dimension(:), allocatable :: n_parameters
        type(evaluator_t), dimension(:), allocatable :: eval
        integer, dimension(:), allocatable :: last_strfun
        integer, dimension(:), allocatable :: out_index
        integer, dimension(:), allocatable :: coll_index
    contains
        (Strfun: strfun chain: TBP)
    end type strfun_chain_t

(Strfun: public)+≡
    public :: strfun_chain_init

(Strfun: procedures)+≡
    subroutine strfun_chain_init (sfchain, beam_data, n_strfun)
        type(strfun_chain_t), intent(out) :: sfchain
        type(beam_data_t), intent(in), target :: beam_data
        integer, intent(in) :: n_strfun
        integer :: i
        sfchain%n_strfun = n_strfun
        allocate (sfchain%strfun (n_strfun))
        allocate (sfchain%n_parameters (n_strfun))
        sfchain%n_parameters = 0
        allocate (sfchain%eval (n_strfun))
        call beam_init (sfchain%beam, beam_data)
        allocate (sfchain%last_strfun (beam_data%n))
    end subroutine strfun_chain_init

```

```

allocate (sfchain%out_index (beam_data%n))
allocate (sfchain%coll_index (beam_data%n))
sfchain%last_strfun = 0
do i = 1, size (sfchain%out_index)
    sfchain%out_index(i) = i
    sfchain%coll_index(i) = i
end do
end subroutine strfun_chain_init

```

This is a separate initialization routine for the structure-function mappings.

```

<Strfun: public>+≡
    public :: strfun_chain_allocate_mappings

<Strfun: procedures>+≡
    subroutine strfun_chain_allocate_mappings &
        (sfchain, multichannel, n_mapping, n_channel)
        type(strfun_chain_t), intent(inout) :: sfchain
        logical, intent(in) :: multichannel
        integer, intent(in) :: n_mapping, n_channel
        sfchain%multichannel = multichannel
        sfchain%n_mapping = n_mapping
        allocate (sfchain%sf_mapping (n_mapping, n_channel))
    end subroutine strfun_chain_allocate_mappings

```

Set beam momenta directly without changing anything else.

```

<Strfun: public>+≡
    public :: strfun_chain_set_beam_momenta

<Strfun: procedures>+≡
    subroutine strfun_chain_set_beam_momenta (sfchain, p)
        type(strfun_chain_t), intent(inout) :: sfchain
        type(vector4_t), dimension(:), intent(in) :: p
        call beam_set_momenta (sfchain%beam, p)
    end subroutine strfun_chain_set_beam_momenta

```

```

<Strfun: public>+≡
    public :: strfun_chain_final

<Strfun: procedures>+≡
    subroutine strfun_chain_final (sfchain)
        type(strfun_chain_t), intent(inout) :: sfchain
        call beam_final (sfchain%beam)
        if (allocated (sfchain%strfun)) call strfun_final (sfchain%strfun)
        if (allocated (sfchain%eval)) call evaluator_final (sfchain%eval)
    end subroutine strfun_chain_final

```

## I/O

```

<Strfun: public>+≡
    public :: strfun_chain_write

<Strfun: strfun chain: TBP>≡
    procedure :: write => strfun_chain_write

```



*<Strfun: procedures>+≡*

```

subroutine strfun_chain_write &
    (sfchain, unit, verbose, show_momentum_sum, show_mass)
class(strfun_chain_t), intent(in) :: sfchain
integer, intent(in), optional :: unit
logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
integer :: u, i, ch
logical :: verb
verb = .false.; if (present (verbose)) verb = verbose
u = output_unit (unit); if (u < 0) return
write (u, *) "Structure function chain:"
write (u, *)
call beam_write (sfchain%beam, unit, verbose, show_momentum_sum, show_mass)
if (allocated (sfchain%strfun)) then
    do i = 1, size (sfchain%strfun)
        write (u, *)
        call strfun_write &
            (sfchain%strfun(i), unit, verbose, show_momentum_sum, show_mass)
        write (u, *) "number of parameters = ", sfchain%n_parameters(i)
    end do
end if
if (allocated (sfchain%sf_mapping)) then
    if (sfchain%multichannel) then
        do ch = 1, size (sfchain%sf_mapping, 2)
            write (u, *)
            write (u, *) "Mappings for channel #", ch
            do i = 1, size (sfchain%sf_mapping, 1)
                call strfun_mapping_write (sfchain%sf_mapping(i, ch), unit)
            end do
        end do
    else
        do i = 1, size (sfchain%sf_mapping, 1)
            write (u, *)
            call strfun_mapping_write (sfchain%sf_mapping(i,1), unit)
        end do
    end if
end if
if (allocated (sfchain%eval)) then
    write (u, *)
    write (u, *) "Evaluators:"
    do i = 1, size (sfchain%eval)
        call evaluator_write &
            (sfchain%eval(i), unit, verbose, show_momentum_sum, show_mass)
    end do
end if
write (u, *)
write (u, *) "Total number of parameters      = ", &
    sfchain%n_parameters_tot
write (u, "(1x,A)", advance="no") "Last structure function (index) = "
if (allocated (sfchain%last_strfun)) then
    write (u, *) sfchain%last_strfun
else
    write (u, *) "[not allocated]"
end if
end if

```

```

write (u, "(1x,A)", advance="no") "Outgoing particles (index)      = "
if (allocated (sfchain%out_index)) then
  write (u, *) sfchain%out_index
else
  write (u, *) "[not allocated]"
end if
write (u, "(1x,A)", advance="no") "Colliding particles (index)    = "
if (allocated (sfchain%coll_index)) then
  write (u, *) sfchain%coll_index
else
  write (u, *) "[not allocated]"
end if
end subroutine strfun_chain_write

```

### Defined assignment

Deep copy of all components.

```

<Strfun: public>+≡
  public :: assignment(=)

<Strfun: interfaces>+≡
  interface assignment(=)
    module procedure strfun_chain_assign
  end interface

<Strfun: procedures>+≡
  subroutine strfun_chain_assign (sfchain_out, sfchain_in)
    type(strfun_chain_t), intent(out) :: sfchain_out
    type(strfun_chain_t), intent(in)  :: sfchain_in
    sfchain_out%beam = sfchain_in%beam
    sfchain_out%n_strfun = sfchain_in%n_strfun
    sfchain_out%multichannel = sfchain_in%multichannel
    sfchain_out%n_mapping = sfchain_in%n_mapping
    if (allocated (sfchain_in%strfun)) then
      allocate (sfchain_out%strfun (size (sfchain_in%strfun)))
      sfchain_out%strfun = sfchain_in%strfun
    end if
    if (allocated (sfchain_in%sf_mapping)) then
      allocate (sfchain_out%sf_mapping &
        (size (sfchain_in%sf_mapping, 1), size (sfchain_in%sf_mapping, 2)))
      sfchain_out%sf_mapping = sfchain_in%sf_mapping
    end if
    sfchain_out%mapping_factor = sfchain_in%mapping_factor
    sfchain_out%n_parameters_tot = sfchain_in%n_parameters_tot
    if (allocated (sfchain_in%n_parameters)) then
      allocate (sfchain_out%n_parameters (size (sfchain_in%n_parameters)))
      sfchain_out%n_parameters = sfchain_in%n_parameters
    end if
    if (allocated (sfchain_in%eval)) then
      allocate (sfchain_out%eval (size (sfchain_in%eval)))
      sfchain_out%eval = sfchain_in%eval
    end if
    if (allocated (sfchain_in%last_strfun)) then

```

```

        allocate (sfchain_out%last_strfun (size (sfchain_in%last_strfun)))
        sfchain_out%last_strfun = sfchain_in%last_strfun
    end if
    if (allocated (sfchain_in%out_index)) then
        allocate (sfchain_out%out_index (size (sfchain_in%out_index)))
        sfchain_out%out_index = sfchain_in%out_index
    end if
    if (allocated (sfchain_in%coll_index)) then
        allocate (sfchain_out%coll_index (size (sfchain_in%coll_index)))
        sfchain_out%coll_index = sfchain_in%coll_index
    end if
end subroutine strfun_chain_assign

```

### Accessing contents

Return the type of PDF used. So far, we only return a type if there are exactly two structure functions and they are both the same.

```

<Strfun: public>+≡
    public :: strfun_chain_get_strfun_type

<Strfun: procedures>+≡
    function strfun_chain_get_strfun_type(sfchain) result(type)
        type(strfun_chain_t), intent(in) :: sfchain
        integer :: type

        if(size(sfchain%strfun).eq.2) then
            if(sfchain%strfun(1)%type .eq. sfchain%strfun(2)%type) then
                type = sfchain%strfun(1)%type
            else
                type = STRF_NONE
            end if
        else
            type = STRF_NONE
        end if
    end function strfun_chain_get_strfun_type

```

Return the number of the member of the PDF used

```

<Strfun: public>+≡
    public :: strfun_chain_get_strfun_set

<Strfun: procedures>+≡
    !! !! function strfun_chain_get_strfun_set(sfchain) result(set)
    !! !!     type(strfun_chain_t), intent(in) :: sfchain
    !! !!     integer :: set
    !! !!
    !! !!     set = 0
    !! !!     if(size(sfchain%strfun).eq.2) then
    !! !!         if(sfchain%strfun(1)%type .eq. sfchain%strfun(2)%type) then
    !! !!             if(sfchain%strfun(1)%type .eq. STRF_LHAPDF) then
    !! !!                 set = lhpdf_data_get_set(sfchain%strfun(1)%lhpdf_data(1))
    !! !!             else if(sfchain%strfun(1)%type .eq. STRF_PDF_BUILTIN) then
    !! !!                 set = pdf_builtin_get_id(sfchain%strfun(1)%pdf_builtin_data(1))
    !! !!             end if
    !! !!         end if
    !! !!     end if

```

```

!! !! end if
!! !! end function strfun_chain_get_strfun_set

```

The number of active structure functions.

```

<Strfun: public>+≡
public :: strfun_chain_get_n_strfun

<Strfun: procedures>+≡
function strfun_chain_get_n_strfun (sfchain) result (n)
integer :: n
type(strfun_chain_t), intent(in) :: sfchain
n = sfchain%n_strfun
end function strfun_chain_get_n_strfun

```

The number of free parameters ( $x$  values) needed for evaluating the structure functions.

```

<Strfun: public>+≡
public :: strfun_chain_get_n_parameters_tot

<Strfun: procedures>+≡
function strfun_chain_get_n_parameters_tot (sfchain) result (n)
integer :: n
type(strfun_chain_t), intent(in) :: sfchain
n = sfchain%n_parameters_tot
end function strfun_chain_get_n_parameters_tot

```

The number of virtual particles in the structure function evaluators. These are the beam particles plus all particles that do not appear as outgoing.

```

<Strfun: public>+≡
public :: strfun_chain_get_n_vir

<Strfun: procedures>+≡
function strfun_chain_get_n_vir (sfchain) result (n)
integer :: n
type(strfun_chain_t), intent(in) :: sfchain
if (sfchain%n_strfun /= 0) then
n = evaluator_get_n_vir (sfchain%eval(sfchain%n_strfun))
else
n = 0
end if
end function strfun_chain_get_n_vir

```

Tell if the structure-function mappings should be applied to individual phase-space channels, or globally.

```

<Strfun: public>+≡
public :: strfun_chain_multichannel_enabled

<Strfun: procedures>+≡
function strfun_chain_multichannel_enabled (sfchain) result (flag)
logical :: flag
type(strfun_chain_t), intent(in) :: sfchain
flag = sfchain%multichannel
end function strfun_chain_multichannel_enabled

```

Return any extra factor resulting from explicit mappings of the  $x$  parameters.

```

(Strfun: public)+≡
    public :: strfun_chain_get_mapping_factor

(Strfun: procedures)+≡
    function strfun_chain_get_mapping_factor (sfchain) result (f)
        real(default) :: f
        type(strfun_chain_t), intent(in) :: sfchain
        f = sfchain%mapping_factor
    end function strfun_chain_get_mapping_factor

```

For pseudo-structure functions that actually are an external generator, the integration region must not be mapped and stay rigid. This function returns an array which tells, for each integration parameter, whether the corresponding integration dimension is rigid.

```

(Strfun: public)+≡
    public :: strfun_chain_dimension_is_rigid

(Strfun: procedures)+≡
    function strfun_chain_dimension_is_rigid (sfchain) result (rigid)
        logical, dimension(:), allocatable :: rigid
        type(strfun_chain_t), intent(in) :: sfchain
        integer :: i, j, k
        allocate (rigid (sfchain%n_parameters_tot))
        k = 0
        do i = 1, size (sfchain%n_parameters)
            do j = 1, sfchain%n_parameters(i)
                k = k + 1
                select case (sfchain%strfun(i)%type)
                case default
                    rigid(k) = .false.
                end select
            end do
        end do
    end function strfun_chain_dimension_is_rigid

```

Return the indices of the colliding particles.

```

(Strfun: public)+≡
    public :: strfun_chain_get_colliding_particles

(Strfun: procedures)+≡
    function strfun_chain_get_colliding_particles (sfchain) result (index)
        integer, dimension(:), allocatable :: index
        type(strfun_chain_t), intent(in) :: sfchain
        allocate (index (size (sfchain%coll_index)))
        index = sfchain%coll_index
    end function strfun_chain_get_colliding_particles

```

Return the quantum-numbers mask for the colliding particles. This is extracted from the last evaluator in the chain.

```

(Strfun: public)+≡
    public :: strfun_chain_get_colliding_particles_mask

```

```

(Strfun: procedures)+≡
function strfun_chain_get_colliding_particles_mask (sfchain) result (mask)
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
  type(strfun_chain_t), intent(in), target :: sfchain
  integer :: n_strfun
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_eval
  allocate (mask (size (sfchain%coll_index)))
  n_strfun = sfchain%n_strfun
  if (n_strfun /= 0) then
    allocate (mask_eval (evaluator_get_n_tot (sfchain%eval(n_strfun))))
    mask_eval = evaluator_get_mask (sfchain%eval(n_strfun))
    mask = mask_eval(sfchain%coll_index)
  else
    mask = interaction_get_mask (beam_get_int_ptr (sfchain%beam))
  end if
end function strfun_chain_get_colliding_particles_mask

```

Return a pointer to the beam interaction.

```

(Strfun: public)+≡
public :: strfun_chain_get_beam_int_ptr

(Strfun: procedures)+≡
function strfun_chain_get_beam_int_ptr (sfchain) result (int)
  type(interaction_t), pointer :: int
  type(strfun_chain_t), intent(in), target :: sfchain
  int => beam_get_int_ptr (sfchain%beam)
end function strfun_chain_get_beam_int_ptr

```

Return a pointer to the last evaluator, which wraps up all structure functions.

```

(Strfun: public)+≡
public :: strfun_chain_get_last_evaluator_ptr

(Strfun: procedures)+≡
function strfun_chain_get_last_evaluator_ptr (sfchain) result (eval)
  type(evaluator_t), pointer :: eval
  type(strfun_chain_t), intent(in), target :: sfchain
  if (sfchain%n_strfun /= 0) then
    eval => sfchain%eval(sfchain%n_strfun)
  else
    eval => null ()
  end if
end function strfun_chain_get_last_evaluator_ptr

```

### Setting up structure functions

The index *i* is the overall structure function counter. *line* indicates the beam(s) for which the structure function applies, either 1 or 2, or 0 for both beams.

```

(Strfun: public)+≡
!! !! public :: strfun_chain_set_strfun

```

*<Strfun: interfaces>+≡*

```
!! !! interface strfun_chain_set_strfun
!! !!   module procedure strfun_chain_set_lhapdf
!! !!   module procedure strfun_chain_set_pdf_builtin
!! !!   module procedure strfun_chain_set_isr
!! !!   module procedure strfun_chain_set_epa
!! !!   module procedure strfun_chain_set_ewa
!! !!   module procedure strfun_chain_set_circe1
!! !!   module procedure strfun_chain_set_circe2
!! !!   module procedure strfun_chain_set_escan
!! !!   module procedure strfun_chain_set_beam_events
!! !!   module procedure strfun_chain_set_user
!! !! end interface
```

*<Strfun: procedures>+≡*

```
!! !! subroutine strfun_chain_set_lhapdf &
!! !!   (sfchain, i, line, lhpdf_data, n_parameters)
!! !!   type(strfun_chain_t), intent(inout), target :: sfchain
!! !!   integer, intent(in) :: i, line, n_parameters
!! !!   type(lhapdf_data_t), intent(in) :: lhpdf_data
!! !!   call strfun_init (sfchain%strfun(i), lhpdf_data)
!! !!   sfchain%n_parameters(i) = n_parameters
!! !!   call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
!! !! end subroutine strfun_chain_set_lhapdf
!! !!
!! !! subroutine strfun_chain_set_pdf_builtin &
!! !!   (sfchain, i, line, pdf_builtin_data, n_parameters)
!! !!   type(strfun_chain_t), intent(inout), target :: sfchain
!! !!   integer, intent(in) :: i, line, n_parameters
!! !!   type(pdf_builtin_data_t), intent(in) :: pdf_builtin_data
!! !!   call strfun_init (sfchain%strfun(i), pdf_builtin_data)
!! !!   sfchain%n_parameters(i) = n_parameters
!! !!   call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
!! !! end subroutine strfun_chain_set_pdf_builtin
!! !!
!! !! subroutine strfun_chain_set_isr &
!! !!   (sfchain, i, line, isr_data, n_parameters)
!! !!   type(strfun_chain_t), intent(inout), target :: sfchain
!! !!   integer, intent(in) :: i, line, n_parameters
!! !!   type(isr_data_t), intent(in) :: isr_data
!! !!   call strfun_init (sfchain%strfun(i), isr_data)
!! !!   sfchain%n_parameters(i) = n_parameters
!! !!   call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
!! !! end subroutine strfun_chain_set_isr
!! !!
!! !! subroutine strfun_chain_set_epa &
!! !!   (sfchain, i, line, epa_data, n_parameters)
!! !!   type(strfun_chain_t), intent(inout), target :: sfchain
!! !!   integer, intent(in) :: i, line, n_parameters
!! !!   type(epa_data_t), intent(in) :: epa_data
!! !!   call strfun_init (sfchain%strfun(i), epa_data)
!! !!   sfchain%n_parameters(i) = n_parameters
!! !!   call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
!! !! end subroutine strfun_chain_set_epa
!! !!
```

```

!! !! subroutine strfun_chain_set_ewa &
!! !!     (sfchain, i, line, ewa_data, n_parameters, id)
!! !!     type(strfun_chain_t), intent(inout), target :: sfchain
!! !!     integer, intent(in) :: i, line, n_parameters, id
!! !!     type(ewa_data_t), intent(inout) :: ewa_data
!! !!     call strfun_init (sfchain%strfun(i), ewa_data, id)
!! !!     sfchain%n_parameters(i) = n_parameters
!! !!     call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
!! !! end subroutine strfun_chain_set_ewa
!! !!
!! !! subroutine strfun_chain_set_circe1 &
!! !!     (sfchain, i, line, circe1_data, n_parameters)
!! !!     type(strfun_chain_t), intent(inout), target :: sfchain
!! !!     integer, intent(in) :: i, line, n_parameters
!! !!     type(circe1_data_t), intent(in) :: circe1_data
!! !!     call strfun_init (sfchain%strfun(i), circe1_data)
!! !!     sfchain%n_parameters(i) = n_parameters
!! !!     call strfun_chain_link (sfchain, i, line, (/1, 2/), (/5, 6/))
!! !! end subroutine strfun_chain_set_circe1
!! !!
!! !! subroutine strfun_chain_set_circe2 &
!! !!     (sfchain, i, line, circe2_data, n_parameters)
!! !!     type(strfun_chain_t), intent(inout), target :: sfchain
!! !!     integer, intent(in) :: i, line, n_parameters
!! !!     type(circe2_data_t), intent(in) :: circe2_data
!! !!     call strfun_init (sfchain%strfun(i), circe2_data)
!! !!     sfchain%n_parameters(i) = n_parameters
!! !!     call strfun_chain_link (sfchain, i, line, (/1, 2/), (/3, 4/))
!! !! end subroutine strfun_chain_set_circe2
!! !!
!! !! subroutine strfun_chain_set_escan &
!! !!     (sfchain, i, line, escan_data, n_parameters)
!! !!     type(strfun_chain_t), intent(inout), target :: sfchain
!! !!     integer, intent(in) :: i, line, n_parameters
!! !!     type(escan_data_t), intent(in) :: escan_data
!! !!     call strfun_init (sfchain%strfun(i), escan_data)
!! !!     sfchain%n_parameters(i) = n_parameters
!! !!     if (line == 0) then
!! !!         call strfun_chain_link (sfchain, i, line, (/1, 2/), (/3, 4/))
!! !!     else
!! !!         call strfun_chain_link (sfchain, i, line, (/1/), (/2/))
!! !!     end if
!! !! end subroutine strfun_chain_set_escan
!! !!
!! !! subroutine strfun_chain_set_beam_events &
!! !!     (sfchain, i, line, beam_events_data, n_parameters)
!! !!     type(strfun_chain_t), intent(inout), target :: sfchain
!! !!     integer, intent(in) :: i, line, n_parameters
!! !!     type(beam_events_data_t), intent(in) :: beam_events_data
!! !!     call strfun_init (sfchain%strfun(i), beam_events_data)
!! !!     sfchain%n_parameters(i) = n_parameters
!! !!     if (line == 0) then
!! !!         call strfun_chain_link (sfchain, i, line, (/1, 2/), (/3, 4/))
!! !!     else

```



```

!! !!      call strfun_chain_link (sfchain, i, line, (/1/), (/2/))
!! !!      end if
!! !! end subroutine strfun_chain_set_beam_events
!! !!
!! !! subroutine strfun_chain_set_user &
!! !!      (sfchain, i, line, user_data, n_parameters)
!! !!      type(strfun_chain_t), intent(inout), target :: sfchain
!! !!      integer, intent(in) :: i, line, n_parameters
!! !!      type(sf_user_data_t), intent(in) :: user_data
!! !!      integer :: n_tot
!! !!      call strfun_init (sfchain%strfun(i), user_data)
!! !!      n_tot = sf_user_data_get_n_tot (user_data)
!! !!      sfchain%n_parameters(i) = n_parameters
!! !!      if (line == 0) then
!! !!          call strfun_chain_link (sfchain, i, line, (/1, 2/), (/n_tot-1, n_tot/))
!! !!      else
!! !!          call strfun_chain_link (sfchain, i, line, (/1/), (/n_tot/))
!! !!      end if
!! !! end subroutine strfun_chain_set_user

```

This procedure links a new structure function to the existing chain. *i* is the overall structure function counter. *line* indicates the beam(s) to which the structure function applies; 0 is for both beams. The last two arguments are the indices of the incoming and outgoing particle(s) within the current structure function. For a single-beam (double-beam) structure function, these arrays are of length 1 (2), respectively.

The connections of outgoing/incoming particles are recorded as links in the new structure-function entry `sfchain%strfun(i)`.

*(Strfun: procedures)*+≡

```

subroutine strfun_chain_link (sfchain, i, line, in_index, out_index)
    type(strfun_chain_t), intent(inout), target :: sfchain
    integer, intent(in) :: i, line
    integer, dimension(:), intent(in) :: in_index, out_index
    select case (line)
    case (0)
        call link_single (1, in_index(1))
        call link_single (2, in_index(2))
        sfchain%last_strfun = i
        sfchain%out_index = out_index
    case default
        call link_single (line, in_index(1))
        sfchain%last_strfun(line) = i
        sfchain%out_index(line) = out_index(1)
    end select
contains
    subroutine link_single (line, in_index)
        integer, intent(in) :: line, in_index
        integer :: j
        j = sfchain%last_strfun(line)
        select case (j)
        case (0)
            call interaction_set_source_link &
                (sfchain%strfun(i)%int, in_index, &

```

```

        sfchain%beam, sfchain%out_index(line))
    case default
        call interaction_set_source_link &
            (sfchain%strfun(i)%int, in_index, &
             sfchain%strfun(j)%int, sfchain%out_index(line))
    end select
end subroutine link_single
end subroutine strfun_chain_link

```

## Setting up mappings

Set a particular mapping with a known type.

```

<Strfun: public>+≡
    public :: strfun_chain_set_mapping

<Strfun: procedures>+≡
    subroutine strfun_chain_set_mapping (sfchain, i, ch, index, type, par)
        type(strfun_chain_t), intent(inout) :: sfchain
        integer, intent(in) :: i, ch
        integer, dimension(:), intent(in) :: index
        integer, intent(in) :: type
        real(default), dimension(:), intent(in) :: par
        call strfun_mapping_init (sfchain%sf_mapping(i, ch), index, type, par)
    end subroutine strfun_chain_set_mapping

```

## Evaluators

```

<Strfun: public>+≡
    public :: strfun_chain_make_evaluators

<Strfun: procedures>+≡
    subroutine strfun_chain_make_evaluators (sfchain, ok)
        type(strfun_chain_t), intent(inout), target :: sfchain
        logical, intent(out), optional :: ok
        type(interaction_t), pointer :: beam_int, eval_int, sf_int, eval_int_next
        type(quantum_numbers_mask_t) :: qn_mask_conn
        type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_beam
        integer :: i, j, last, out_index, coll_index
        sfchain%n_parameters_tot = sum (sfchain%n_parameters)
        beam_int => beam_get_int_ptr (sfchain%beam)
        if (.not. associated (beam_int)) call msg_bug &
            ("strfun_chain_make_evaluators: null beam pointer")
        allocate (qn_mask_beam (interaction_get_n_out (beam_int)))
        qn_mask_beam = interaction_get_mask (beam_int)
        call interaction_exchange_mask (beam_int)
        do i = 1, size (sfchain%strfun) - 1
            call interaction_exchange_mask (sfchain%strfun(i)%int)
        end do
        do i = size (sfchain%strfun), 1, -1
            call interaction_exchange_mask (sfchain%strfun(i)%int)
        end do
        if (any (qn_mask_beam .neqv. interaction_get_mask (beam_int))) then

```

```

        call beam_write (sfchain%beam)
        call msg_fatal (" Beam polarization/color/flavor incompatible with structure functions")
    end if
    eval_int => beam_int
    do i = 1, size (sfchain%strfun)
        qn_mask_conn = new_quantum_numbers_mask (.false., .false., .true.)
        call evaluator_init_product (sfchain%eval(i), eval_int, &
            sfchain%strfun(i)%int, qn_mask_conn)
        if (evaluator_is_empty (sfchain%eval(i))) then
            call msg_fatal ("Mismatch in beam and structure-function chain")
            if (present (ok)) ok = .false.
            return
        end if
        eval_int => evaluator_get_int_ptr (sfchain%eval(i))
    end do
    if (size (sfchain%strfun) /= 0) then
        do j = 1, size (sfchain%coll_index)
            last = sfchain%last_strfun(j)
            select case (last)
            case (0)
                eval_int => beam_get_int_ptr (sfchain%beam)
                out_index = sfchain%out_index(j)
                coll_index = out_index
            case default
                sf_int => sfchain%strfun(last)%int
                eval_int => evaluator_get_int_ptr (sfchain%eval(last))
                out_index = sfchain%out_index(j)
                coll_index = interaction_find_link (eval_int, sf_int, out_index)
            end select
            if (coll_index /= 0) then
                do i = last + 1, size (sfchain%strfun)
                    out_index = coll_index
                    eval_int_next => evaluator_get_int_ptr (sfchain%eval(i))
                    coll_index = &
                        interaction_find_link (eval_int_next, eval_int, out_index)
                    if (coll_index == 0) call msg_bug ("Structure functions: " &
                        // "broken links in structure function chain")
                    eval_int => eval_int_next
                end do
            end if
            if (coll_index /= 0) then
                sfchain%coll_index(j) = coll_index
            else
                call msg_bug ("Structure functions: " &
                    // "colliding particles can't be determined")
            end if
        end do
    end if
    if (present (ok)) ok = .true.
end subroutine strfun_chain_make_evaluators

```

Setup kinematics: use the given array of random numbers `r` to generate a chain of momenta, up to the incoming partons.

```

<Strfun: public>+=
  public :: strfun_chain_set_kinematics

<Strfun: procedures>+=
  subroutine strfun_chain_set_kinematics (sfchain, r, &
    channel, offset, r_all, sf_factor, ok)
    type(strfun_chain_t), intent(inout) :: sfchain
    real(default), dimension(:), intent(in) :: r
    integer, intent(in), optional :: channel, offset
    real(default), dimension(:,:), intent(inout), optional :: r_all
    real(default), dimension(:), intent(out), optional :: sf_factor
    logical, intent(out), optional :: ok
    real(default), dimension(size(r)) :: x
    integer :: n_mapping
    integer :: i, i1, i2, ch, n, n1, n_sf
    real(default) :: xprod, factor
    real(default), dimension(:), allocatable :: factor_channel
    integer, dimension(:), allocatable :: mapping_type
    n_sf = size (sfchain%strfun)
    sfchain%mapping_factor = 1
    if (size (r) == sfchain%n_parameters_tot) then
      x = r
      if (allocated (sfchain%sf_mapping)) then
        n_mapping = size (sfchain%sf_mapping, 1)
        if (present (channel)) then
          allocate (factor_channel (n_mapping))
          allocate (mapping_type (n_mapping))
          do i = 1, n_mapping
            mapping_type(i) = sfchain%sf_mapping(i,channel)%type
            call strfun_mapping_apply &
              (sfchain%sf_mapping(i, channel), x, factor_channel(i))
          end do
          sf_factor(channel) = product (factor_channel)
        else
          do i = 1, n_mapping
            call strfun_mapping_apply &
              (sfchain%sf_mapping(i, 1), x, factor)
            sfchain%mapping_factor = sfchain%mapping_factor * factor
          end do
        end if
      end if
      n = 0
      do i = 1, size (sfchain%strfun)
        call interaction_receive_momenta (sfchain%strfun(i)%int)
        n1 = sfchain%n_parameters(i)
        call strfun_set_kinematics (sfchain%strfun(i), x(n+1:n+n1), .false.)
        n = n + n1
      end do
      if (present (channel)) then
        i1 = offset
        i2 = offset + size (r)
        do ch = 1, size (r_all, 2)
          if (ch /= channel) then
            sf_factor(ch) = 1
            do i = 1, n_mapping

```

```

        if (sfchain%sf_mapping(i,ch)%type == mapping_type(i)) then
            r_all(i1+1:i2,ch) = r
            sf_factor(ch) = sf_factor(ch) * factor_channel(i)
        else
            r_all(i1+1:i2,ch) = x
            call strfun_mapping_apply_inverse &
                (sfchain%sf_mapping(1,ch), r_all(i1+1:i2,ch), &
                 factor)
            sf_factor(ch) = sf_factor(ch) * factor
        end if
    end do
end if
end do
end if
do i = 1, size (sfchain%strfun)
    call evaluator_receive_momenta (sfchain%eval(i))
end do
if (present (ok)) ok = .true.
else
    call msg_bug ("Structure functions: mismatch in number of parameters")
end if
end subroutine strfun_chain_set_kinematics

```

*<Strfun: public>+≡*

```
public :: strfun_chain_evaluate
```

*<Strfun: procedures>+≡*

```

subroutine strfun_chain_evaluate (sfchain, scale)
    type(strfun_chain_t), intent(inout) :: sfchain
    real(default), intent(in) :: scale
    integer :: i
    do i = size (sfchain%strfun), 1, -1
        call strfun_apply (sfchain%strfun(i), scale)
    end do
    do i = 1, size (sfchain%eval)
        call evaluator_evaluate (sfchain%eval(i))
    end do
end subroutine strfun_chain_evaluate

```

## 10.15.4 Test

*<Strfun: public>+≡*

```
public :: strfun_test
```

*<Strfun: procedures>+≡*

```

!! !! subroutine strfun_test (lhpdf_present)
!! !! use os_interface, only: os_data_t
!! !! type(os_data_t) :: os_data
!! !! type(model_t), pointer :: model
!! !! logical, intent(in) :: lhpdf_present
!! !! print *, "*** Read model file"
!! !! call syntax_model_file_init ()
!! !! call model_list_read_model &

```

```

!! !!      (var_str("SM"), var_str("SM.mdl"), os_data, model)
!! !!      call syntax_model_file_final ()
!! !!      print *, "*****"
!! !!      call isr_test (model)
!! !!      print *, "*****"
!! !!      call epa_test (model)
!! !!      if (lhpdf_present) then
!! !!          print *, "*****"
!! !!          call lhpdf_test (model)
!! !!      end if
!! !! end subroutine strfun_test

!! !! subroutine isr_test (model)
!! !!     use flavors
!! !!     use polarizations
!! !!     type(model_t), intent(in), target :: model
!! !!     type(flavor_t), dimension(2) :: flv
!! !!     type(polarization_t), dimension(2) :: pol
!! !!     type(beam_data_t), target :: beam_data
!! !!     type(isr_data_t), dimension(2) :: isr_data
!! !!     type(strfun_chain_t), target :: sfchain
!! !!     integer :: i
!! !!     print *, "*** ISR test"
!! !!     call flavor_init (flv, (/11, -11/), model)
!! !!     call polarization_init_unpolarized (pol(1), flv(1))
!! !!     call polarization_init_unpolarized (pol(2), flv(2))
!! !!     call beam_data_init_sqrts (beam_data, 500._default, flv, pol)
!! !!     do i = 1, 2
!! !!         call isr_data_init (isr_data(i), &
!! !!             model, flv(i), 0.06_default, 500._default, 0.511e-3_default)
!! !!     end do
!! !!     call strfun_chain_init (sfchain, beam_data, 2)
!! !!     call strfun_chain_set_strfun (sfchain, 1, 1, isr_data(1), 1)
!! !!     call strfun_chain_set_strfun (sfchain, 2, 2, isr_data(2), 3)
!! !!     call strfun_chain_make_evaluators (sfchain)
!! !!     call strfun_chain_set_kinematics &
!! !!         (sfchain, (/0.8_default, 0.4_default, 0.5_default, 0.2_default/))
!! !!     call strfun_chain_evaluate (sfchain, 0._default)
!! !!     call strfun_chain_write (sfchain)
!! !!     call strfun_chain_final (sfchain)
!! !! end subroutine isr_test

subroutine epa_test (model)
    use flavors
    use polarizations
    type(model_t), intent(in), target :: model
    type(flavor_t), dimension(2) :: flv
    type(polarization_t), dimension(2) :: pol
    type(beam_data_t) :: beam_data
    type(epa_data_t) :: epa_data1
    type(epa_data_t) :: epa_data2
    type(strfun_chain_t), target :: sfchain
    print *, "*** EPA test"
    call flavor_init (flv, (/2, 1/), model)

```

```

! Prepare beams
call polarization_init_circular (pol(1), flv(1), 0.3_default)
call polarization_init_unpolarized (pol(2), flv(2))
call beam_data_init_sqrts (beam_data, 1000._default, flv, pol)
call strfun_chain_init (sfchain, beam_data, 2)
! Initialize EPA for both
call epa_data_init (epa_data1, model, &
    flv(1), 0.06_default, 1.e-6_default, 0._default, 500._default, &
    511.e-6_default)
call epa_data_init (epa_data2, model, &
    flv(2), 0.06_default, 1.e-6_default, 1._default, 500._default)
call strfun_chain_set_strfun (sfchain, 1, 1, epa_data1, 1)
call strfun_chain_set_strfun (sfchain, 2, 2, epa_data2, 3)
! call strfun_chain_write (sfchain); stop
call strfun_chain_make_evaluators (sfchain)
call strfun_chain_set_kinematics &
    (sfchain, (/0.8_default, 0.4_default, 0.5_default, 0.2_default/))
call strfun_chain_evaluate (sfchain, 0._default)
call strfun_chain_write (sfchain)
! Clean up
call beam_data_final (beam_data)
call polarization_final (pol)
call strfun_chain_final (sfchain)
end subroutine epa_test

!! !! subroutine lhpdf_test (model)
!! !! use flavors
!! !! use polarizations
!! !! type(model_t), intent(in), target :: model
!! !! type(beam_data_t) :: beam_data
!! !! type(flavor_t), dimension(2) :: flv
!! !! type(polarization_t), dimension(2) :: pol
!! !! type(lhpdf_data_t), dimension(2) :: data
!! !! type(lhpdf_status_t) :: lhpdf_status
!! !! type(strfun_chain_t), target :: sfchain
!! !! real(default) :: scale
!! !! print *, "*** LHAPDF test"
!! !! call flavor_init (flv, (/ -PROTON, PHOTON /), model)
!! !! call polarization_init_unpolarized (pol(1), flv(1))
!! !! call polarization_init_unpolarized (pol(2), flv(2))
!! !! call beam_data_init_sqrts (beam_data, 2000._default, flv, pol)
!! !! call strfun_chain_init (sfchain, beam_data, 2)
!! !! call lhpdf_data_init (data(1), lhpdf_status, model, flv(1), member=1)
!! !! !!! Use the same photon PDF that is demanded by the LHAPDF tests.
!! !! call lhpdf_data_init (data(2), lhpdf_status, model, flv(2), &
!! !! file=var_str("GSG961.LHgrid"), photon_scheme=1)
!! !! call lhpdf_data_set_mask (data(2), &
!! !! (/ .false., .false., .false., .true., .true., .true., &
!! !! .false., &
!! !! .true., .true., .true., .false., .false., .false. /))
!! !! call strfun_chain_write (sfchain); stop
!! !! call strfun_chain_set_strfun (sfchain, 1, 1, data(1), 1)
!! !! call strfun_chain_set_strfun (sfchain, 2, 2, data(2), 1)
!! !! call strfun_chain_write (sfchain); stop

```

```
!! !! call strfun_chain_make_evaluators (sfchain)
!! !! call strfun_chain_set_kinematics (sfchain, (/0.9_default, 0.4_default/))
!! !! scale = 1.e3_default
!! !! call strfun_chain_evaluate (sfchain, scale)
!! !! call strfun_chain_write (sfchain)
!! !! call strfun_chain_final (sfchain)
!! !! end subroutine lhpdf_test
```



## Chapter 11

# Phase space and hard matrix elements

These modules contain the internal representation and evaluation of phase space and the interface to (hard-)process evaluation.

**mappings** Generate invariant masses and decay angles from given random numbers (or the inverse operation). Each mapping pertains to a particular node in a phase-space tree. Different mappings account for uniform distributions, resonances, zero-mass behavior, and so on.

**phs\_trees** Phase space parameterizations for scattering processes are defined recursively as if there was an initial particle decaying. This module sets up a representation in terms of abstract trees, where each node gets a unique binary number. Each tree is stored as an array of branches, where integers indicate the connections. This emulates pointers in a transparent way. Real pointers would also be possible, but seem to be less efficient for this particular case.

**phs\_forests** The type defined by this module collects the decay trees corresponding to a given process and the applicable mappings. To set this up, a file is read which is either written by the user or by the **cascades** module functions. The module also contains the routines that evaluate phase space, i.e., generate momenta from random numbers and back.

**cascades** This module is a Feynman diagram generator with the particular purpose of finding the phase space parameterizations best suited for a given process. It uses a model file to set up the possible vertices, generates all possible diagrams, identifies resonances and singularities, and simplifies the list by merging equivalent diagrams and dropping irrelevant ones. This process can be controlled at several points by user-defined parameters. Note that it depends on the particular values of particle masses, so it cannot be done before reading the input file.

## 11.1 Process data block

We define a simple transparent type that contains universal constant process data. We will reference objects of this type for the phase-space setup, for interfacing with process libraries, for implementing matrix-element generation, and in the master process-handling module.

```
<process_constants.f90>≡  
  <File header>
```

```
  module process_constants
```

```
    <Use kinds>
```

```
    <Use strings>
```

```
    <Standard module head>
```

```
    <Process constants: public>
```

```
    <Process constants: types>
```

```
  end module process_constants
```

The data type is just a block of public objects, only elementary types, no type-bound procedures.

```
<Process constants: public>≡
```

```
  public :: process_constants_t
```

```
<Process constants: types>≡
```

```
  type :: process_constants_t
```

```
    type(string_t) :: id
```

```
    type(string_t) :: model_name
```

```
    character(32) :: md5sum = ""
```

```
    logical :: openmp_supported = .false.
```

```
    integer :: n_in = 0
```

```
    integer :: n_out = 0
```

```
    integer :: n_flv = 0
```

```
    integer :: n_hel = 0
```

```
    integer :: n_col = 0
```

```
    integer :: n_cin = 0
```

```
    integer :: n_cf = 0
```

```
    integer, dimension(:,:), allocatable :: flv_state
```

```
    integer, dimension(:,:), allocatable :: hel_state
```

```
    integer, dimension(:,:), allocatable :: col_state
```

```
    logical, dimension(:,:), allocatable :: ghost_flag
```

```
    complex(default), dimension(:), allocatable :: color_factors
```

```
    integer, dimension(:,:), allocatable :: cf_index
```

```
  end type process_constants_t
```

## 11.2 Abstract phase-space module

In this module we define an abstract base type (and a trivial test implementation) for multi-channel phase-space parameterizations.

```
<phs_base.f90>≡  
<File header>  
  
module phs_base  
  
  <Use kinds>  
  <Use strings>  
  <Use file utils>  
  use constants !NODEP!  
  use diagnostics !NODEP!  
  use lorentz !NODEP!  
  use unit_tests  
  use os_interface  
  use md5  
  use variables  
  use models  
  use flavors  
  use process_constants  
  use sf_mappings  
  use sf_base  
  
  <Standard module head>  
  
  <PHS base: public>  
  
  <PHS base: types>  
  
  <PHS base: interfaces>  
  
  <PHS base: test types>  
  
  contains  
  
  <PHS base: procedures>  
  
  <PHS base: tests>  
  
end module phs_base
```

### 11.2.1 Phase-space channels

The kinematics configuration may generate multiple parameterizations of phase space. Some of those have specific properties, such as a resonance in the s channel.

This is the abstract type for the channel properties.

```
<PHS base: types>≡  
  type, abstract :: channel_prop_t  
  contains  
    procedure (channel_prop_to_string), deferred :: to_string
```

```
end type channel_prop_t
```

```
<PHS base: interfaces>≡
abstract interface
  function channel_prop_to_string (object) result (string)
    import
    class(channel_prop_t), intent(in) :: object
    type(string_t) :: string
  end function channel_prop_to_string
end interface
```

This type describes an equivalence. The current channel is equivalent to channel c. The equivalence involves a permutation **perm** of integration dimensions and, within each integration dimension, a mapping **mode**.

```
<PHS base: types>+≡
type :: phs_equivalence_t
  integer :: c = 0
  integer, dimension(:), allocatable :: perm
  integer, dimension(:), allocatable :: mode
contains
  <PHS base: phs equivalence: TBP>
end type phs_equivalence_t
```

The mapping modes are

```
<PHS base: types>+≡
integer, parameter, public :: &
  EQ_IDENTITY = 0, EQ_INVERT = 1, EQ_SYMMETRIC = 2, EQ_INVARIANT = 3
```

In particular, if a channel is equivalent to itself in the EQ\_SYMMETRIC mode, the integrand can be assumed to be symmetric w.r.t. a reflection  $x \rightarrow 1 - x$  of the corresponding integration variable.

These are the associated tags, for output:

```
<PHS base: types>+≡
character, dimension(0:3), parameter :: TAG = ["+", "-", ":", "x"]
```

Write an equivalence.

```
<PHS base: phs equivalence: TBP>≡
procedure :: write => phs_equivalence_write

<PHS base: procedures>≡
subroutine phs_equivalence_write (object, unit)
  class(phs_equivalence_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, j
  u = output_unit (unit)
  write (u, "(5x,'=',1x,I0,1x)", advance = "no") object%c
  if (allocated (object%perm)) then
    write (u, "(A)", advance = "no") "("
    do j = 1, size (object%perm)
      if (j > 1) write (u, "(1x)", advance = "no")
      write (u, "(I0,A1)", advance = "no") &
```

```

            object%perm(j), TAG(object%mode(j))
        end do
        write (u, "(A)" " ")
    else
        write (u, "(A)")
    end if
end subroutine phs_equivalence_write

```

Initialize an equivalence. This allocates the `perm` and `mode` arrays with equal size.

```

<PHS base: phs equivalence: TBP>+=
    procedure :: init => phs_equivalence_init

<PHS base: procedures>+=
    subroutine phs_equivalence_init (eq, n_dim)
        class(phs_equivalence_t), intent(out) :: eq
        integer, intent(in) :: n_dim
        allocate (eq%perm (n_dim), source = 0)
        allocate (eq%mode (n_dim), source = EQ_IDENTITY)
    end subroutine phs_equivalence_init

```

The channel entry holds (optionally) specific properties.

`sf_channel` is the structure-function channel that corresponds to this phase-space channel. The structure-function channel may be set up with a specific mapping that depends on the phase-space channel properties. (The default setting is to leave the properties empty.)

```

<PHS base: public>=
    public :: phs_channel_t

<PHS base: types>+=
    type :: phs_channel_t
        class(channel_prop_t), allocatable :: prop
        integer :: sf_channel = 1
        type(phs_equivalence_t), dimension(:), allocatable :: eq
    contains
        <PHS base: phs channel: TBP>
    end type phs_channel_t

```

Output.

```

<PHS base: phs channel: TBP>=
    procedure :: write => phs_channel_write

<PHS base: procedures>+=
    subroutine phs_channel_write (object, unit)
        class(phs_channel_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u, j
        u = output_unit (unit)
        write (u, "(1x,I0)", advance="no") object%sf_channel
        if (allocated (object%prop)) then
            write (u, "(1x,A)" char (object%prop%to_string ()))
        else
            write (u, *)
        end if
    end subroutine phs_channel_write

```

```

end if
if (allocated (object%eq)) then
  do j = 1, size (object%eq)
    call object%eq(j)%write (u)
  end do
end if
end subroutine phs_channel_write

```

## 11.2.2 Kinematics configuration

Here, we store the universal information that is specifically relevant for phase-space generation. It is a subset of the process data, supplemented by basic information on phase-space parameterization channels.

A concrete implementation will contain more data, that describe the phase space in detail.

```

<PHS base: public>+≡
  public :: phs_config_t

<PHS base: types>+≡
  type, abstract :: phs_config_t
  !
    private
      type(string_t) :: id
      integer :: n_in = 0
      integer :: n_out = 0
      integer :: n_tot = 0
      integer :: n_state = 0
      integer :: n_par = 0
      integer :: n_channel = 0
      real(default) :: sqrts = 0
      logical :: sqrts_fixed = .true.
      logical :: cm_frame = .true.
      logical :: azimuthal_dependence = .false.
      integer, dimension(:), allocatable :: dim_flat
      logical :: provides_equivalences = .false.
      logical :: provides_chains = .false.
      integer, dimension(:), allocatable :: chain
      type(model_t), pointer :: model => null ()
      type(flavor_t), dimension(:, :), allocatable :: flv
      type(phs_channel_t), dimension(:), allocatable :: channel
      character(32) :: md5sum_process = ""
      character(32) :: md5sum_model_par = ""
      character(32) :: md5sum_phs_config = ""
    contains
      <PHS base: phs config: TBP>
  end type phs_config_t

```

Finalizer, deferred.

```

<PHS base: phs config: TBP>≡
  procedure (phs_config_final), deferred :: final

```

```

<PHS base: interfaces>+=
  abstract interface
    subroutine phs_config_final (object)
      import
      class(phs_config_t), intent(inout) :: object
    end subroutine phs_config_final
  end interface

```

Output. We provide an implementation for the output of the base-type contents and an interface for the actual write method.

```

<PHS base: phs config: TBP>+=
  procedure (phs_config_write), deferred :: write
  procedure :: base_write => phs_config_write

<PHS base: procedures>+=
  subroutine phs_config_write (object, unit)
    class(phs_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i, j
    u = output_unit (unit)
    write (u, "(3x,A,A,A)") "ID          = '", char (object%id), "'
    write (u, "(3x,A,I0)")  "n_in       = ", object%n_in
    write (u, "(3x,A,I0)")  "n_out      = ", object%n_out
    write (u, "(3x,A,I0)")  "n_tot      = ", object%n_tot
    write (u, "(3x,A,I0)")  "n_state    = ", object%n_state
    write (u, "(3x,A,I0)")  "n_par      = ", object%n_par
    write (u, "(3x,A,I0)")  "n_channel = ", object%n_channel
    write (u, "(3x,A,ES19.12)") "sqrts    = ", object%sqrts
    write (u, "(3x,A,L1)")  "s_fixed   = ", object%sqrts_fixed
    write (u, "(3x,A,L1)")  "cm_frame  = ", object%cm_frame
    write (u, "(3x,A,L1)")  "azim.dep. = ", object%azimuthal_dependence
    if (allocated (object%dim_flat)) then
      write (u, "(3x,A,I0)")  "flat dim. = ", object%dim_flat
    end if
    write (u, "(1x,A)")  "Flavor combinations:"
    do i = 1, object%n_state
      write (u, "(3x,I0,':')", advance="no") i
      do j = 1, object%n_tot
        write (u, "(1x,A)", advance="no") &
          char (flavor_get_name (object%flv(j,i)))
      end do
      write (u, *)
    end do
    if (allocated (object%channel)) then
      write (u, "(1x,A)")  "Phase-space / structure-function channels:"
      do i = 1, object%n_channel
        write (u, "(3x,I0,':')", advance="no") i
        call object%channel(i)%write (u)
      end do
    end if
    if (object%md5sum_process /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (process)    = '", &
        object%md5sum_process, "'"
    end if

```

```

if (object%md5sum_model_par /= "") then
  write (u, "(3x,A,A,A)") "MD5 sum (model par) = ', &
    object%md5sum_model_par, "'"
end if
if (object%md5sum_phs_config /= "") then
  write (u, "(3x,A,A,A)") "MD5 sum (phs config) = ', &
    object%md5sum_phs_config, "'"
end if
end subroutine phs_config_write

```

Similarly, a basic initializer and an interface. We assume that the model file has been read, so we can just fetch a model pointer.

The intent is `inout`. We want to be able to set parameters in advance.

*(PHS base: phs config: TBP)+≡*

```

procedure :: init => phs_config_init

```

*(PHS base: procedures)+≡*

```

subroutine phs_config_init (phs_config, data)
  class(phs_config_t), intent(inout) :: phs_config
  type(process_constants_t), intent(in) :: data
  integer :: i, j
  phs_config%id = data%id
  phs_config%n_in = data%n_in
  phs_config%n_out = data%n_out
  phs_config%n_tot = data%n_in + data%n_out
  phs_config%n_state = data%n_flv
  phs_config%model => model_list_get_model_ptr (data%model_name)
  allocate (phs_config%flv (phs_config%n_tot, phs_config%n_state))
  do i = 1, phs_config%n_state
    do j = 1, phs_config%n_tot
      call flavor_init (phs_config%flv(j,i), data%flv_state(j,i), &
        phs_config%model)
    end do
  end do
  phs_config%md5sum_process = data%md5sum
end subroutine phs_config_init

```

This procedure should complete the phase-space configuration. We need the `sqrts` value as overall scale, which is known only after the beams have been defined. The procedure should determine the number of channels, their properties (if any), and allocate and fill the `channel` array accordingly.

*(PHS base: phs config: TBP)+≡*

```

procedure (phs_config_configure), deferred :: configure

```

*(PHS base: interfaces)+≡*

```

abstract interface
  subroutine phs_config_configure (phs_config, sqrts, &
    sqrts_fixed, cm_frame, azimuthal_dependence, rebuild)
  import
  class(phs_config_t), intent(inout) :: phs_config
  real(default), intent(in) :: sqrts
  logical, intent(in), optional :: sqrts_fixed
  logical, intent(in), optional :: cm_frame
end interface

```



```

        logical, intent(in), optional :: azimuthal_dependence
        logical, intent(in), optional :: rebuild
    end subroutine phs_config_configure
end interface

```

Compute the MD5 sum. We abuse the `write` method. In type implementations, `write` should only display information that is relevant for the MD5 sum. The data include the process MD5 sum which is taken from the process constants, and the MD5 sum of the model parameters. This may change, so it is computed here.

```

<PHS base: phs config: TBP>+≡
    procedure :: compute_md5sum => phs_config_compute_md5sum
<PHS base: procedures>+≡
    subroutine phs_config_compute_md5sum (phs_config)
        class(phs_config_t), intent(inout) :: phs_config
        integer :: u
        phs_config%md5sum_model_par = model_get_parameters_md5sum (phs_config%model)
        phs_config%md5sum_phs_config = ""
        u = free_unit ()
        open (u, status = "scratch", action = "readwrite")
        call phs_config%write (u)
        rewind (u)
        phs_config%md5sum_phs_config = md5sum (u)
        close (u)
    end subroutine phs_config_compute_md5sum

```

Print an informative message after phase-space configuration.

```

<PHS base: phs config: TBP>+≡
    procedure (phs_startup_message), deferred :: startup_message
    procedure :: base_startup_message => phs_startup_message
<PHS base: procedures>+≡
    subroutine phs_startup_message (phs_config, unit)
        class(phs_config_t), intent(in) :: phs_config
        integer, intent(in), optional :: unit
        write (msg_buffer, "(A,3(1x,I0,1x,A))") &
            "Phase space:", &
            phs_config%n_channel, "channels,", &
            phs_config%n_par, "dimensions"
        call msg_message (unit = unit)
    end subroutine phs_startup_message

```

This procedure should assign structure-function channels to the phase-space channels. The assignment will depend on the phase-space channel properties (which are known at this point) and on the available structure-function channels or mappings.

For the actual implementation, this procedure has to apply heuristics as to which structure-function mappings are appropriate for which phase-space structure. The mappings have to be prepared in advance. This requires extracting information (e.g., about resonance parameters) from the configured phase space.

```

<PHS base: phs config: TBP>+≡
    procedure (phs_config_match_channels), deferred :: match_channels

```

```

<PHS base: interfaces>+=
  abstract interface
    subroutine phs_config_match_channels (phs_config, sf, sf_channel)
      import
      class(phs_config_t), intent(inout) :: phs_config
      type(sf_config_t), dimension(:), intent(in), allocatable :: sf
      type(sf_channel_t), dimension(:), intent(in), allocatable :: sf_channel
    end subroutine phs_config_match_channels
  end interface

```

This procedure should be implemented such that the phase-space configuration object allocates a phase-space instance of matching type.

```

<PHS base: phs config: TBP>+=
  procedure (phs_config_allocate_instance), nopass, deferred :: &
    allocate_instance

```

```

<PHS base: interfaces>+=
  abstract interface
    subroutine phs_config_allocate_instance (phs)
      import
      class(phs_t), intent(inout), pointer :: phs
    end subroutine phs_config_allocate_instance
  end interface

```

### 11.2.3 Extract data

Return the number of MC input parameters.

```

<PHS base: phs config: TBP>+=
  procedure :: get_n_par => phs_config_get_n_par

<PHS base: procedures>+=
  function phs_config_get_n_par (phs_config) result (n)
    class(phs_config_t), intent(in) :: phs_config
    integer :: n
    n = phs_config%n_par
  end function phs_config_get_n_par

```

Return dimensions (parameter indices) for which the phase-space dimension is flat, so integration and event generation can be simplified.

```

<PHS base: phs config: TBP>+=
  procedure :: get_flat_dimensions => phs_config_get_flat_dimensions

<PHS base: procedures>+=
  function phs_config_get_flat_dimensions (phs_config) result (dim_flat)
    class(phs_config_t), intent(in) :: phs_config
    integer, dimension(:), allocatable :: dim_flat
    if (allocated (phs_config%dim_flat)) then
      allocate (dim_flat (size (phs_config%dim_flat)))
      dim_flat = phs_config%dim_flat
    else
      allocate (dim_flat (0))
    end if
  end function

```

```
end function phs_config_get_flat_dimensions
```

Return the number of phase-space channels.

```
<PHS base: phs config: TBP>+≡
  procedure :: get_n_channel => phs_config_get_n_channel

<PHS base: procedures>+≡
  function phs_config_get_n_channel (phs_config) result (n)
    class(phs_config_t), intent(in) :: phs_config
    integer :: n
    n = phs_config%n_channel
  end function phs_config_get_n_channel
```

Return the structure-function channel that corresponds to the phase-space channel *c*. If the channel array is not allocated (which happens if there is no structure function), return zero.

```
<PHS base: phs config: TBP>+≡
  procedure :: get_sf_channel => phs_config_get_sf_channel

<PHS base: procedures>+≡
  function phs_config_get_sf_channel (phs_config, c) result (c_sf)
    class(phs_config_t), intent(in) :: phs_config
    integer, intent(in) :: c
    integer :: c_sf
    if (allocated (phs_config%channel)) then
      c_sf = phs_config%channel(c)%sf_channel
    else
      c_sf = 0
    end if
  end function phs_config_get_sf_channel
```

Return the mass(es) of the incoming particle(s). We take the first flavor combination in the array, assuming that masses must be degenerate among flavors.

```
<PHS base: phs config: TBP>+≡
  procedure :: get_masses_in => phs_config_get_masses_in

<PHS base: procedures>+≡
  subroutine phs_config_get_masses_in (phs_config, m)
    class(phs_config_t), intent(in) :: phs_config
    real(default), dimension(:), intent(out) :: m
    integer :: i
    do i = 1, phs_config%n_in
      m(i) = flavor_get_mass (phs_config%flv(i,1))
    end do
  end subroutine phs_config_get_masses_in
```

Return the MD5 sum of the configuration.

```
<PHS base: phs config: TBP>+≡
  procedure :: get_md5sum => phs_config_get_md5sum
```

```

(PHS base: procedures)+≡
function phs_config_get_md5sum (phs_config) result (md5sum)
  class(phs_config_t), intent(in) :: phs_config
  character(32) :: md5sum
  md5sum = phs_config%md5sum_phs_config
end function phs_config_get_md5sum

```

#### 11.2.4 Phase-space point instance

The `phs_t` object holds the workspace for phase-space generation. In the base object, we have the MC input parameters `r` and the Jacobian factor `f`, for each channel, and the incoming and outgoing momenta.

Note: The `active_channel` array is not used yet, all elements are initialized with `.true.`. It should be touched by the integrator if it decides to drop irrelevant channels.

```

(PHS base: public)+≡
public :: phs_t

(PHS base: types)+≡
type, abstract :: phs_t
  class(phs_config_t), pointer :: config => null ()
  logical :: r_defined = .false.
  integer :: selected_channel = 0
  logical, dimension(:), allocatable :: active_channel
  real(default), dimension(:,:), allocatable :: r
  real(default), dimension(:), allocatable :: f
  real(default), dimension(:), allocatable :: m_in
  real(default), dimension(:), allocatable :: m_out
  real(default) :: flux = 0
  real(default) :: volume = 0
  type(lorentz_transformation_t) :: lt_cm_to_lab
  logical :: p_defined = .false.
  real(default) :: sqrts_hat = 0
  type(vector4_t), dimension(:), allocatable :: p
  logical :: q_defined = .false.
  type(vector4_t), dimension(:), allocatable :: q
contains
  (PHS base: phs: TBP)
end type phs_t

```

Output. Since phase space may get complicated, we include a `verbose` option for the abstract `write` procedure.

```

(PHS base: phs: TBP)≡
procedure (phs_write), deferred :: write

(PHS base: interfaces)+≡
abstract interface
  subroutine phs_write (object, unit, verbose)
    import
    class(phs_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
  end subroutine

```

```

        end subroutine phs_write
    end interface

```

This procedure can be called to print the contents of the base type.

```

<PHS base: phs: TBP>+≡
    procedure :: base_write => phs_base_write

<PHS base: procedures>+≡
    subroutine phs_base_write (object, unit)
        class(phs_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u, c, i
        u = output_unit (unit)
        write (u, "(1x,A)", advance="no") "Partonic phase space: parameters"
        if (object%r_defined) then
            write (u, *)
        else
            write (u, "(1x,A)") "[undefined]"
        end if
        write (u, "(3x,A,999(1x,ES19.12))") "m_in   =", object%m_in
        write (u, "(3x,A,999(1x,ES19.12))") "m_out   =", object%m_out
        write (u, "(3x,A,ES19.12)") "Flux    =", object%flux
        write (u, "(3x,A,ES19.12)") "Volume  =", object%volume
        if (allocated (object%f)) then
            do c = 1, size (object%r, 2)
                write (u, "(1x,A,IO,A)", advance="no") "Channel #", c, ":"
                if (c == object%selected_channel) then
                    write (u, "(1x,A)") "[selected]"
                else
                    write (u, *)
                end if
                write (u, "(3x,A)", advance="no") "r ="
                do i = 1, size (object%r, 1)
                    write (u, "(1x,F9.7)", advance="no") object%r(i,c)
                end do
                write (u, *)
                write (u, "(3x,A,1x,ES13.7)") "f =", object%f(c)
            end do
        end if
        write (u, "(1x,A)") "Partonic phase space: momenta"
        if (object%p_defined) then
            write (u, "(3x,A,ES19.12)") "sqrts  =", object%sqrts_hat
        end if
        write (u, "(1x,A)", advance="no") "Incoming:"
        if (object%p_defined) then
            write (u, *)
        else
            write (u, "(1x,A)") "[undefined]"
        end if
        if (allocated (object%p)) then
            do i = 1, size (object%p)
                call vector4_write (object%p(i), u)
            end do
        end if
    end subroutine

```

```

write (u, "(1x,A)", advance="no") "Outgoing:"
if (object%q_defined) then
  write (u, *)
else
  write (u, "(1x,A)") "[undefined]"
end if
if (allocated (object%q)) then
  do i = 1, size (object%q)
    call vector4_write (object%q(i), u)
  end do
end if
if (object%p_defined .and. .not. object%config%cm_frame) then
  write (u, "(1x,A)") "Transformation c.m -> lab frame"
  call lorentz_transformation_write (object%lt_cm_to_lab, u)
end if
end subroutine phs_base_write

```

Finalizer. The base type does not need it, but extensions may.

```

<PHS base: phs: TBP>+≡
  procedure (phs_final), deferred :: final

<PHS base: interfaces>+≡
  abstract interface
    subroutine phs_final (object)
      import
      class(phs_t), intent(inout) :: object
    end subroutine phs_final
  end interface

```

Initializer. Everything should be contained in the `process_data` configuration object, so we can require a universal interface.

```

<PHS base: phs: TBP>+≡
  procedure (phs_init), deferred :: init

<PHS base: interfaces>+≡
  abstract interface
    subroutine phs_init (phs, phs_config)
      import
      class(phs_t), intent(out) :: phs
      class(phs_config_t), intent(in), target :: phs_config
    end subroutine phs_init
  end interface

```

The base version will just allocate the arrays. It should be called at the beginning of the implementation of `phs_init`.

```

<PHS base: phs: TBP>+≡
  procedure :: base_init => phs_base_init

<PHS base: procedures>+≡
  subroutine phs_base_init (phs, phs_config)
    class(phs_t), intent(out) :: phs
    class(phs_config_t), intent(in), target :: phs_config
    phs%config => phs_config
  end subroutine

```

```

allocate (phs%active_channel (phs%config%n_channel))
phs%active_channel = .true.
allocate (phs%r (phs%config%n_par, phs%config%n_channel)); phs%r = 0
allocate (phs%f (phs%config%n_channel)); phs%f = 0
allocate (phs%p (phs%config%n_in))
allocate (phs%q (phs%config%n_out))
allocate (phs%m_in (phs%config%n_in), &
          source = flavor_get_mass (phs_config%flv(:phs_config%n_in, 1)))
allocate (phs%m_out (phs%config%n_out), &
          source = flavor_get_mass (phs_config%flv(phs_config%n_in+1:, 1)))
call phs%compute_flux ()
end subroutine phs_base_init

```

Manually select a channel.

```

(PHS base: phs: TBP)+≡
  procedure :: select_channel => phs_base_select_channel

(PHS base: procedures)+≡
  subroutine phs_base_select_channel (phs, channel)
    class(phs_t), intent(inout) :: phs
    integer, intent(in), optional :: channel
    if (present (channel)) then
      phs%selected_channel = channel
    else
      phs%selected_channel = 0
    end if
  end subroutine phs_base_select_channel

```

Set incoming momenta. Assume that array shapes match. If requested, compute the Lorentz transformation from the c.m. to the lab frame and apply that transformation to the incoming momenta.

```

(PHS base: phs: TBP)+≡
  procedure :: set_incoming_momenta => phs_set_incoming_momenta

(PHS base: procedures)+≡
  subroutine phs_set_incoming_momenta (phs, p)
    class(phs_t), intent(inout) :: phs
    type(vector4_t), dimension(:), intent(in) :: p
    type(vector4_t) :: p0, p1
    type(lorentz_transformation_t) :: lt0
    phs%p = p
    if (phs%config%cm_frame) then
      phs%sqrts_hat = phs%config%sqrts
      phs%p = p
    else
      p0 = sum (p)
      if (phs%config%sqrts_fixed) then
        phs%sqrts_hat = phs%config%sqrts
      else
        phs%sqrts_hat = p0 ** 1
      end if
      lt0 = boost (p0, phs%sqrts_hat)
      p1 = inverse (lt0) * p(1)
      phs%lt_cm_to_lab = lt0 * rotation_to_2nd (3, space_part (p1))
    end if
  end subroutine phs_set_incoming_momenta

```

```

        phs%p = inverse (phs%lt_cm_to_lab) * p
    end if
    phs%p_defined = .true.
end subroutine phs_set_incoming_momenta

```

Set outgoing momenta. Assume that array shapes match. The incoming momenta must be known, so can apply the Lorentz transformation from c.m. to lab (inverse) to the momenta.

```

(PHS base: phs: TBP)+≡
    procedure :: set_outgoing_momenta => phs_set_outgoing_momenta
(PHS base: procedures)+≡
    subroutine phs_set_outgoing_momenta (phs, q)
        class(phs_t), intent(inout) :: phs
        type(vector4_t), dimension(:), intent(in) :: q
        if (phs%p_defined) then
            if (phs%config%cm_frame) then
                phs%q = q
            else
                phs%q = inverse (phs%lt_cm_to_lab) * q
            end if
            phs%q_defined = .true.
        end if
    end subroutine phs_set_outgoing_momenta

```

Return outgoing momenta. Apply the c.m. to lab transformation if necessary.

```

(PHS base: phs: TBP)+≡
    procedure :: get_outgoing_momenta => phs_get_outgoing_momenta
(PHS base: procedures)+≡
    subroutine phs_get_outgoing_momenta (phs, q)
        class(phs_t), intent(in) :: phs
        type(vector4_t), dimension(:), intent(out) :: q
        if (phs%p_defined .and. phs%q_defined) then
            if (phs%config%cm_frame) then
                q = phs%q
            else
                q = phs%lt_cm_to_lab * phs%q
            end if
        else
            q = vector4_null
        end if
    end subroutine phs_get_outgoing_momenta

```

Return the input parameter array for a channel.

```

(PHS base: phs: TBP)+≡
    procedure :: get_mcpair => phs_get_mcpair
(PHS base: procedures)+≡
    subroutine phs_get_mcpair (phs, c, r)
        class(phs_t), intent(in) :: phs
        integer, intent(in) :: c
        real(default), dimension(:), intent(out) :: r

```



```

    if (phs%r_defined) then
        r = phs%r(:,c)
    else
        r = 0
        ! error?
    end if
end subroutine phs_get_mcpair

```

Return the Jacobian factor for a channel.

```

<PHS base: phs: TBP>+≡
    procedure :: get_f => phs_get_f

<PHS base: procedures>+≡
    function phs_get_f (phs, c) result (f)
        class(phs_t), intent(in) :: phs
        integer, intent(in) :: c
        real(default) :: f
        if (phs%r_defined) then
            f = phs%f(c)
        else
            f = 0
            ! error?
        end if
    end function phs_get_f

```

Return the overall factor, which is the product of the flux factor for the incoming partons and the phase-space volume for the outgoing partons.

```

<PHS base: phs: TBP>+≡
    procedure :: get_overall_factor => phs_get_overall_factor

<PHS base: procedures>+≡
    function phs_get_overall_factor (phs) result (f)
        class(phs_t), intent(in) :: phs
        real(default) :: f
        f = phs%flux * phs%volume
    end function phs_get_overall_factor

```

Compute flux factor. We do this during initialization (when the incoming momenta  $p$  are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

```

<PHS base: phs: TBP>+≡
    procedure :: compute_flux => phs_compute_flux

<PHS base: procedures>+≡
    subroutine phs_compute_flux (phs)
        class(phs_t), intent(inout) :: phs
        real(default) :: s_hat, lda
        select case (phs%config%n_in)
        case (1)
            if (.not. phs%p_defined) then
                phs%flux = twopi4 / (2 * phs%m_in(1))
            end if
        case (2)

```

```

    if (phs%p_defined) then
      if (phs%config%sqrts_fixed) then
        return
      else
        s_hat = sum (phs%p) ** 2
      end if
    else
      if (phs%config%sqrts_fixed) then
        s_hat = phs%config%sqrts ** 2
      else
        return
      end if
    end if
    lda = lambda (s_hat, phs%m_in(1) ** 2, phs%m_in(2) ** 2)
    if (lda > 0) then
      phs%flux = conv * twopi4 / (2 * sqrt (lda))
    else
      phs%flux = 0
    end if
  end select
end subroutine phs_compute_flux

```

Evaluate the phase-space point for a particular channel and compute momenta, Jacobian, and phase-space volume. Also compute the inverse mappings to completely fill the **r** and **f** arrays. This is, of course, deferred to the implementation.

```

(PHS base: phs: TBP)+≡
  procedure (phs_evaluate), deferred :: evaluate

(PHS base: interfaces)+≡
  abstract interface
    subroutine phs_evaluate (phs, c_in, r_in)
      import
      class(phs_t), intent(inout) :: phs
      integer, intent(in) :: c_in
      real(default), dimension(:), intent(in) :: r_in
    end subroutine phs_evaluate
  end interface

```

Inverse evaluation. If all momenta are known, we compute the inverse mappings to fill the **r** and **f** arrays.

```

(PHS base: phs: TBP)+≡
  procedure (phs_inverse), deferred :: inverse

(PHS base: interfaces)+≡
  abstract interface
    subroutine phs_inverse (phs)
      import
      class(phs_t), intent(inout) :: phs
    end subroutine phs_inverse
  end interface

```

### 11.2.5 Auxiliary stuff

The `pacify` subroutine, which is provided by the Lorentz module, has the purpose of setting numbers to zero which are (by comparing with a `tolerance` parameter) considered equivalent with zero. This is useful for numerical checks.

```
<PHS base: public>+≡
    public :: pacify

<PHS base: interfaces>+≡
    interface pacify
        module procedure pacify_phs
    end interface pacify

<PHS base: procedures>+≡
    subroutine pacify_phs (phs)
        class(phs_t), intent(inout) :: phs
        if (phs%p_defined) then
            call pacify (phs%p, 10 * epsilon (1._default) * phs%config%sqrts)
            call pacify (phs%lt_cm_to_lab, 10 * epsilon (1._default))
        end if
        if (phs%q_defined) then
            call pacify (phs%q, 10 * epsilon (1._default) * phs%config%sqrts)
        end if
    end subroutine pacify_phs
```

### 11.2.6 Unit tests

```
<PHS base: public>+≡
    public :: phs_base_test

<PHS base: tests>≡
    subroutine phs_base_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <PHS base: execute tests>
    end subroutine phs_base_test
```

### Test process data

We provide a procedure that initializes a test case for the process constants. This set of process data contains just the minimal contents that we need for the phase space. The rest is left uninitialized.

```
<PHS base: public>+≡
    public :: init_test_process_data

<PHS base: tests>+≡
    subroutine init_test_process_data (id, data)
        type(process_constants_t), intent(out) :: data
        type(string_t), intent(in), optional :: id
        if (present (id)) then
            data%id = id
        else
```

```

        data%id = "testproc"
    end if
    data%model_name = "Test"
    data%n_in = 2
    data%n_out = 2
    data%n_flv = 1
    allocate (data%flv_state (data%n_in + data%n_out, data%n_flv))
    data%flv_state = 25
end subroutine init_test_process_data

```

### Test kinematics configuration

This is a trivial implementation of the `phs_config_t` configuration object.

```

<PHS base: public>+≡
    public :: phs_test_config_t

<PHS base: test types>≡
    type, extends (phs_config_t) :: phs_test_config_t
        logical :: create_equivalences = .false.
    contains
        procedure :: final => phs_test_config_final
        procedure :: write => phs_test_config_write
        procedure :: configure => phs_test_config_configure
        procedure :: startup_message => phs_test_config_startup_message
        procedure :: match_channels => phs_test_config_match_channels
        procedure, nopass :: allocate_instance => phs_test_config_allocate_instance
    end type phs_test_config_t

```

The finalizer is empty.

```

<PHS base: tests>+≡
    subroutine phs_test_config_final (object)
        class(phs_test_config_t), intent(inout) :: object
    end subroutine phs_test_config_final

```

The `cm_frame` parameter is not tested here; we defer this to the `phs_single` implementation.

```

<PHS base: tests>+≡
    subroutine phs_test_config_write (object, unit)
        class(phs_test_config_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "Partonic phase-space configuration:"
        call object%base_write (unit)
    end subroutine phs_test_config_write

    subroutine phs_test_config_configure &
        (phs_config, sqrts, sqrts_fixed, cm_frame, azimuthal_dependence, rebuild)
        class(phs_test_config_t), intent(inout) :: phs_config
        real(default), intent(in) :: sqrts
        logical, intent(in), optional :: sqrts_fixed
        logical, intent(in), optional :: cm_frame
    end subroutine phs_test_config_configure

```

```

logical, intent(in), optional :: azimuthal_dependence
logical, intent(in), optional :: rebuild
phs_config%n_channel = 2
phs_config%n_par = 2
phs_config%sqrts = sqrts
if (present (sqrts_fixed)) then
    phs_config%sqrts_fixed = sqrts_fixed
end if
if (present (cm_frame)) then
    phs_config%cm_frame = cm_frame
end if
if (present (azimuthal_dependence)) then
    phs_config%azimuthal_dependence = azimuthal_dependence
end if
if (allocated (phs_config%channel)) deallocate (phs_config%channel)
allocate (phs_config%channel (phs_config%n_channel))
if (phs_config%create_equivalences) then
    call setup_test_equivalences (phs_config)
end if
call phs_config%compute_md5sum ()
end subroutine phs_test_config_configure

```

If requested, we make up an arbitrary set of equivalences.

*(PHS base: tests)*+≡

```

subroutine setup_test_equivalences (phs_config)
class(phs_test_config_t), intent(inout) :: phs_config
integer :: i
associate (channel => phs_config%channel(1))
    allocate (channel%eq (2))
    do i = 1, size (channel%eq)
        call channel%eq(i)%init (phs_config%n_par)
    end do
    associate (eq => channel%eq(1))
        eq%c = 1; eq%perm = [1, 2]; eq%mode = [EQ_IDENTITY, EQ_SYMMETRIC]
    end associate
    associate (eq => channel%eq(2))
        eq%c = 2; eq%perm = [2, 1]; eq%mode = [EQ_INVARIANT, EQ_IDENTITY]
    end associate
end associate
end subroutine setup_test_equivalences

```

Startup message

*(PHS base: tests)*+≡

```

subroutine phs_test_config_startup_message (phs_config, unit)
class(phs_test_config_t), intent(in) :: phs_config
integer, intent(in), optional :: unit
call phs_config%base_startup_message (unit)
write (msg_buffer, "(A)") "Phase space: Test"
call msg_message (unit = unit)
end subroutine phs_test_config_startup_message

```

This method has the duty of matching phase-space channels with structure-function channels.

For the test implementation, we just check the number of structure-function channels. If there are two, we assign the channels one-by-one.

```

<PHS base: tests>+≡
subroutine phs_test_config_match_channels (phs_config, sf, sf_channel)
class(phs_test_config_t), intent(inout) :: phs_config
type(sf_config_t), dimension(:), intent(in), allocatable :: sf
type(sf_channel_t), dimension(:), intent(in), allocatable :: sf_channel
if (allocated (sf_channel)) then
  select case (size (sf_channel))
  case (1)
    phs_config%channel%sf_channel = 1
  case (2)
    phs_config%channel(1)%sf_channel = 1
    phs_config%channel(2)%sf_channel = 2
  end select
end if
end subroutine phs_test_config_match_channels

```

The instance type that matches `phs_test_config_t` is `phs_test_t`.

```

<PHS base: tests>+≡
subroutine phs_test_config_allocate_instance (phs)
class(phs_t), intent(inout), pointer :: phs
allocate (phs_test_t :: phs)
end subroutine phs_test_config_allocate_instance

```

## Test kinematics implementation

This implementation of kinematics generates a simple two-particle configuration from the incoming momenta. The incoming momenta must be in the c.m. system, all masses equal.

There are two channels: one generates  $\cos \theta$  and  $\phi$  uniformly, in the other channel we map the  $r_1$  parameter which belongs to  $\cos \theta$ .

We should store the mass parameter that we need.

```

<PHS base: public>+≡
public :: phs_test_t

<PHS base: test types>+≡
type, extends (phs_t) :: phs_test_t
  real(default) :: m = 0
contains
  <PHS base: phs test: TBP>
end type phs_test_t

```

Output. The specific data are displayed only if `verbose` is set.

```

<PHS base: phs test: TBP>≡
procedure :: write => phs_test_write

<PHS base: tests>+≡
subroutine phs_test_write (object, unit, verbose)
class(phs_test_t), intent(in) :: object
integer, intent(in), optional :: unit

```

```

logical, intent(in), optional :: verbose
integer :: u
logical :: verb
u = output_unit (unit)
verb = .false.; if (present (verbose)) verb = verbose
if (verb) then
    write (u, "(1x,A)") "Partonic phase space: data"
    write (u, "(3x,A,ES19.12)") "m = ", object%m
end if
call object%base_write (u)
end subroutine phs_test_write

```

The finalizer is empty.

```

<PHS base: phs test: TBP>+≡
    procedure :: final => phs_test_final

<PHS base: tests>+≡
    subroutine phs_test_final (object)
        class(phs_test_t), intent(inout) :: object
    end subroutine phs_test_final

```

Initialization: set the mass value.

```

<PHS base: phs test: TBP>+≡
    procedure :: init => phs_test_init

<PHS base: tests>+≡
    subroutine phs_test_init (phs, phs_config)
        class(phs_test_t), intent(out) :: phs
        class(phs_config_t), intent(in), target :: phs_config
        call phs%base_init (phs_config)
        phs%m = flavor_get_mass (phs%config%flv(1,1))
    end subroutine phs_test_init

```

Evaluation. In channel 1, we uniformly generate  $\cos\theta$  and  $\phi$ , with Jacobian normalized to one. In channel 2, we prepend a mapping  $r_1 \rightarrow r_1^{(1/3)}$  with Jacobian  $f = 3r_1^2$ .

```

<PHS base: phs test: TBP>+≡
    procedure :: evaluate => phs_test_evaluate

<PHS base: tests>+≡
    subroutine phs_test_evaluate (phs, c_in, r_in)
        class(phs_test_t), intent(inout) :: phs
        integer, intent(in) :: c_in
        real(default), intent(in), dimension(:) :: r_in
        integer :: c, n_channel
        real(default), dimension(:), allocatable :: x
        if (phs%p_defined) then
            call phs%select_channel (c_in)
            n_channel = phs%config%n_channel
            allocate (x (phs%config%n_par))
            phs%r(:,c_in) = r_in
            select case (c_in)
            case (1)

```

```

        x = r_in
    case (2)
        x(1) = r_in(1) ** (1 / 3._default)
        x(2) = r_in(2)
    end select
    call compute_kinematics_solid_angle (phs%p, phs%q, x)
    do c = 1, n_channel
        if (c /= c_in) then
            call inverse_kinematics_solid_angle (phs%p, phs%q, x)
            select case (c)
            case (1)
                phs%r(:,c) = x
            case (2)
                phs%r(1,c) = x(1) ** 3
                phs%r(2,c) = x(2)
            end select
        end if
    end do
    phs%f(1) = 1
    if (phs%r(1,2) /= 0) then
        phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
    else
        phs%f(2) = 0
    end if
    phs%volume = 1
    phs%q_defined = .true.
    phs%r_defined = .true.
end if
end subroutine phs_test_evaluate

```

Inverse evaluation.

```

<PHS base: phs test: TBP>+≡
    procedure :: inverse => phs_test_inverse

<PHS base: tests>+≡
    subroutine phs_test_inverse (phs)
        class(phs_test_t), intent(inout) :: phs
        integer :: c, n_channel
        real(default), dimension(:), allocatable :: x
        if (phs%p_defined .and. phs%q_defined) then
            call phs%select_channel ()
            n_channel = phs%config%n_channel
            allocate (x (phs%config%n_par))
            do c = 1, n_channel
                call inverse_kinematics_solid_angle (phs%p, phs%q, x)
                select case (c)
                case (1)
                    phs%r(:,c) = x
                case (2)
                    phs%r(1,c) = x(1) ** 3
                    phs%r(2,c) = x(2)
                end select
            end do
            phs%f(1) = 1

```



```

      if (phs%r(1,2) /= 0) then
        phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
      else
        phs%f(2) = 0
      end if
      phs%volume = 1
      phs%r_defined = .true.
    end if
  end subroutine phs_test_inverse

```

### Uniform angular distribution

These procedures implement the uniform angular distribution, generated from two parameters  $x_1$  and  $x_2$ :

$$\cos \theta = 1 - 2x_1, \quad \phi = 2\pi x_2 \quad (11.1)$$

We generate a rotation (Lorentz transformation) which rotates the positive  $z$  axis into this point on the unit sphere. This rotation is applied to the incoming momenta, which are assumed to be back-to-back and on-shell.

We do not compute a Jacobian (constant). The uniform distribution is assumed to be normalized.

```

<PHS base: public>+≡
  public :: compute_kinematics_solid_angle

<PHS base: tests>+≡
  subroutine compute_kinematics_solid_angle (p, q, x)
    type(vector4_t), dimension(2), intent(in) :: p
    type(vector4_t), dimension(2), intent(out) :: q
    real(default), dimension(2), intent(in) :: x
    real(default) :: ct, st, phi
    type(lorentz_transformation_t) :: rot
    integer :: i
    ct = 1 - 2*x(1)
    st = sqrt (1 - ct**2)
    phi = twopi * x(2)
    rot = rotation (phi, 3) * rotation (ct, st, 2)
    do i = 1, 2
      q(i) = rot * p(i)
    end do
  end subroutine compute_kinematics_solid_angle

```

This is the inverse transformation. We assume that the outgoing momenta are rotated versions of the incoming momenta, back-to-back. Thus, we determine the angles from  $q(1)$  alone.  $p$  is unused.

```

<PHS base: public>+≡
  public :: inverse_kinematics_solid_angle

<PHS base: tests>+≡
  subroutine inverse_kinematics_solid_angle (p, q, x)
    type(vector4_t), dimension(2), intent(in) :: p
    type(vector4_t), dimension(2), intent(in) :: q

```

```

real(default), dimension(2), intent(out) :: x
real(default) :: ct, phi
ct = polar_angle_ct (q(1))
phi = azimuthal_angle (q(1))
x(1) = (1 - ct) / 2
x(2) = phi / twopi
end subroutine inverse_kinematics_solid_angle

```

## Phase-space configuration data

Construct and display a test phase-space configuration object.

```

<PHS base: execute tests>≡
  call test (phs_base_1, "phs_base_1", &
    "phase-space configuration", &
    u, results)

<PHS base: tests>+≡
  subroutine phs_base_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data

    write (u, "(A)")  "* Test output: phs_base_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
      &test phase-space configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)

    write (u, "(A)")  "* Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_base_1"), process_data)

    allocate (phs_test_config_t :: phs_data)
    call phs_data%init (process_data)

    call phs_data%write (u)

    call phs_data%final ()
    call model_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_base_1"

  end subroutine phs_base_1

```

## Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

*(PHS base: execute tests)*+≡

```
call test (phs_base_2, "phs_base_2", &
  "phase-space evaluation", &
  u, results)
```

*(PHS base: tests)*+≡

```
subroutine phs_base_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  type(process_constants_t) :: process_data
  real(default) :: sqrts, E
  class(phs_config_t), allocatable, target :: phs_data
  class(phs_t), pointer :: phs => null ()
  type(vector4_t), dimension(2) :: p, q

  write (u, "(A)")  "* Test output: phs_base_2"
  write (u, "(A)")  "* Purpose: test simple two-channel phase space"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

  write (u, "(A)")  "* Initialize a process and a matching &
    &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_base_2"), process_data)

  allocate (phs_test_config_t :: phs_data)
  call phs_data%init (process_data)

  sqrts = 1000._default
  call phs_data%configure (sqrts)

  call phs_data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize the phase-space instance"
  write (u, "(A)")

  call phs_data%allocate_instance (phs)
  select type (phs)
  type is (phs_test_t)
    call phs%init (phs_data)
  end select

  call phs%write (u, verbose=.true.)
```

```

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point in channel 1 &
    &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate (1, [0.5_default, 0.125_default])
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point in channel 2 &
    &for x = 0.125, 0.125"
write (u, "(A)")

call phs%evaluate (2, [0.125_default, 0.125_default])
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
select type (phs_data)
type is (phs_test_config_t)
    call phs_data%configure (sqrts)
end select

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call phs%write (u)

call phs%final ()
deallocate (phs)

```

```

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_base_2"

end subroutine phs_base_2

```

## Phase-space equivalences

Construct a test phase-space configuration which contains channel equivalences.

```

<PHS base: execute tests>+≡
  call test (phs_base_3, "phs_base_3", &
    "channel equivalences", &
    u, results)

<PHS base: tests>+≡
  subroutine phs_base_3 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data

    write (u, "(A)")  "* Test output: phs_base_3"
    write (u, "(A)")  "* Purpose: construct phase-space configuration data &
      &with equivalences"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)

    write (u, "(A)")  "* Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_base_3"), process_data)

    allocate (phs_test_config_t :: phs_data)
    call phs_data%init (process_data)
    select type (phs_data)
    type is (phs_test_config_t)
      phs_data%create_equivalences = .true.
    end select

    call phs_data%configure (1000._default)
    call phs_data%write (u)

    call phs_data%final ()
    call model_list_final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_base_3"

end subroutine phs_base_3

```

## MD5 sum checks

Construct a test phase-space configuration, compute and compare MD5 sums.

```

<PHS base: execute tests>+≡
  call test (phs_base_4, "phs_base_4", &
    "MD5 sum", &
    u, results)

<PHS base: tests>+≡
  subroutine phs_base_4 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data
    type(var_list_t), pointer :: var_list

    write (u, "(A)")  "* Test output: phs_base_4"
    write (u, "(A)")  "* Purpose: compute and compare MD5 sums"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)

    write (u, "(A)")  "* Model parameters"
    write (u, "(A)")

    var_list => model_get_var_list_ptr (model)
    call var_list_write (var_list, u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_base_4"), process_data)
    process_data%md5sum = "test_process_data_m6sum_12345678"

    allocate (phs_test_config_t :: phs_data)
    call phs_data%init (process_data)

    call phs_data%compute_md5sum ()
    call phs_data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Modify model parameter"

```

```

write (u, "(A)")

call var_list_set_real (var_list, var_str ("ms"), 100._default, &
    is_known=.true.)
call var_list_write (var_list, u)

write (u, "(A)")
write (u, "(A)")  "* PHS configuration"
write (u, "(A)")

call phs_data%compute_md5sum ()
call phs_data%write (u)

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_base_4"

end subroutine phs_base_4

```

## 11.3 Single-particle phase space

This module implements the phase space for a single particle, i.e., the solid angle, in a straightforward parameterization with a single channel. The phase-space implementation may be used either for  $1 \rightarrow 2$  decays or for  $2 \rightarrow 2$  scattering processes, so the number of incoming particles is the only free parameter in the configuration. In the latter case, we should restrict its use to non-resonant s-channel processes, because there is no mapping of the scattering angle.

(We might extend this later to account for generic  $2 \rightarrow 2$  situations, e.g., account for a Coulomb singularity or detect an s-channel resonance structure that requires matching structure-function mappings.)

This is derived from the `phs_test` implementation in the `phs_base` module above, even more simplified, but intended for actual use.

```
<phs_single.f90>≡  
  <File header>  
  
  module phs_single  
  
    <Use kinds>  
    <Use strings>  
    <Use file utils>  
    use diagnostics !NODEP!  
    use unit_tests  
    use os_interface  
    use constants !NODEP!  
    use lorentz !NODEP!  
    use models  
    use flavors  
    use process_constants  
    use sf_mappings  
    use sf_base  
    use phs_base  
  
    <Standard module head>  
  
    <PHS single: public>  
  
    <PHS single: types>  
  
    contains  
  
    <PHS single: procedures>  
  
    <PHS single: tests>  
  
  end module phs_single
```

### 11.3.1 Configuration

```
<PHS single: public>≡  
  public :: phs_single_config_t
```



```

<PHS single: types>≡
  type, extends (phs_config_t) :: phs_single_config_t
  contains
    <PHS single: phs single config: TBP>
  end type phs_single_config_t

```

The finalizer is empty.

```

<PHS single: phs single config: TBP>≡
  procedure :: final => phs_single_config_final

<PHS single: procedures>≡
  subroutine phs_single_config_final (object)
    class(phs_single_config_t), intent(inout) :: object
  end subroutine phs_single_config_final

```

Output.

```

<PHS single: phs single config: TBP>+≡
  procedure :: write => phs_single_config_write

<PHS single: procedures>+≡
  subroutine phs_single_config_write (object, unit)
    class(phs_single_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Partonic phase-space configuration (single-particle):"
    call object%base_write (unit)
  end subroutine phs_single_config_write

```

Configuration: there is only one channel and two parameters. The second parameter is the azimuthal angle, which may be a flat dimension.

```

<PHS single: phs single config: TBP>+≡
  procedure :: configure => phs_single_config_configure

<PHS single: procedures>+≡
  subroutine phs_single_config_configure &
    (phs_config, sqrts, sqrts_fixed, cm_frame, azimuthal_dependence, rebuild)
    class(phs_single_config_t), intent(inout) :: phs_config
    real(default), intent(in) :: sqrts
    logical, intent(in), optional :: sqrts_fixed
    logical, intent(in), optional :: cm_frame
    logical, intent(in), optional :: azimuthal_dependence
    logical, intent(in), optional :: rebuild
    if (phs_config%n_out == 2) then
      phs_config%n_channel = 1
      phs_config%n_par = 2
      phs_config%sqrts = sqrts
      if (present (sqrts_fixed)) phs_config%sqrts_fixed = sqrts_fixed
      if (present (cm_frame)) phs_config%cm_frame = cm_frame
      if (present (azimuthal_dependence)) then
        phs_config%azimuthal_dependence = azimuthal_dependence
      if (.not. azimuthal_dependence) then
        allocate (phs_config%dim_flat (1))

```

```

        phs_config%dim_flat(1) = 2
    end if
end if
if (allocated (phs_config%channel)) deallocate (phs_config%channel)
allocate (phs_config%channel (1))
call phs_config%compute_md5sum ()
else
    call msg_fatal ("Single-particle phase space requires n_out = 2")
end if
end subroutine phs_single_config_configure

```

Startup message, after configuration is complete.

```

<PHS single: phs single config: TBP>+≡
    procedure :: startup_message => phs_single_config_startup_message

<PHS single: procedures>+≡
    subroutine phs_single_config_startup_message (phs_config, unit)
        class(phs_single_config_t), intent(in) :: phs_config
        integer, intent(in), optional :: unit
        call phs_config%base_startup_message (unit)
        write (msg_buffer, "(A,2(1x,I0,1x,A))") &
            "Phase space: single-particle"
        call msg_message (unit = unit)
    end subroutine phs_single_config_startup_message

```

Match channels: there is only one phase-space channel, which we match with the first structure-function channel.

(Note: we might extend the implementation to account for resonances in the s-channel.)

```

<PHS single: phs single config: TBP>+≡
    procedure :: match_channels => phs_single_config_match_channels

<PHS single: procedures>+≡
    subroutine phs_single_config_match_channels (phs_config, sf, sf_channel)
        class(phs_single_config_t), intent(inout) :: phs_config
        type(sf_config_t), dimension(:), intent(in), allocatable :: sf
        type(sf_channel_t), dimension(:), intent(in), allocatable :: sf_channel
        if (allocated (sf_channel)) then
            select case (size (sf_channel))
            case (1)
                phs_config%channel%sf_channel = 1
            case (2)
                phs_config%channel(1)%sf_channel = 1
                phs_config%channel(2)%sf_channel = 2
            end select
        end if
    end subroutine phs_single_config_match_channels

```

Allocate an instance: the actual phase-space object.

```

<PHS single: phs single config: TBP>+≡
    procedure, nopass :: allocate_instance => phs_single_config_allocate_instance

```

```

<PHS single: procedures>+≡
  subroutine phs_single_config_allocate_instance (phs)
    class(phs_t), intent(inout), pointer :: phs
    allocate (phs_single_t :: phs)
  end subroutine phs_single_config_allocate_instance

```

### 11.3.2 Kinematics implementation

We generate  $\cos\theta$  and  $\phi$  uniformly, covering the solid angle.

Note: The incoming momenta must be in the c.m. system.

```

<PHS single: public>+≡
  public :: phs_single_t

<PHS single: types>+≡
  type, extends (phs_t) :: phs_single_t
  contains
    <PHS single: phs single: TBP>
  end type phs_single_t

```

Output. The `verbose` setting is irrelevant, we just display the contents of the base object.

```

<PHS single: phs single: TBP>≡
  procedure :: write => phs_single_write

<PHS single: procedures>+≡
  subroutine phs_single_write (object, unit, verbose)
    class(phs_single_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    logical :: verb
    u = output_unit (unit)
    verb = .false.; if (present (verbose)) verb = verbose
    call object%base_write (u)
  end subroutine phs_single_write

```

The finalizer is empty.

```

<PHS single: phs single: TBP>+≡
  procedure :: final => phs_single_final

<PHS single: procedures>+≡
  subroutine phs_single_final (object)
    class(phs_single_t), intent(inout) :: object
  end subroutine phs_single_final

```

Initialization. We allocate arrays (`base_init`) and adjust the phase-space volume. The massless two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \quad (11.2)$$

For a decay with nonvanishing masses ( $m_3, m_4$ ), there is a correction factor

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}} \lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \quad (11.3)$$

For a scattering process with nonvanishing masses, the correction factor is

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}^2} \lambda^{1/2}(\hat{s}, m_1^2, m_2^2) \lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \quad (11.4)$$

If the energy is fixed, this is constant. Otherwise, we have to account for varying  $\hat{s}$ .

```

<PHS single: phs single: TBP>+≡
  procedure :: init => phs_single_init
<PHS single: procedures>+≡
  subroutine phs_single_init (phs, phs_config)
    class(phs_single_t), intent(out) :: phs
    class(phs_config_t), intent(in), target :: phs_config
    call phs%base_init (phs_config)
    phs%volume = 1 / (4 * twopi5)
    call phs%compute_factor ()
  end subroutine phs_single_init

```

Compute the correction factor for nonzero masses. We do this during initialization (when the incoming momenta  $\mathbf{p}$  are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

```

<PHS single: phs single: TBP>+≡
  procedure :: compute_factor => phs_single_compute_factor
<PHS single: procedures>+≡
  subroutine phs_single_compute_factor (phs)
    class(phs_single_t), intent(inout) :: phs
    real(default) :: s_hat, lda1, lda2
    select case (phs%config%n_in)
    case (1)
      if (.not. phs%p_defined) then
        if (sum (phs%m_out) < phs%m_in(1)) then
          s_hat = phs%m_in(1) ** 2
          phs%f(1) = 1 / s_hat &
            * sqrt (lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2))
        else
          phs%f(1) = 0
        end if
      end if
    case (2)
      if (phs%config%sqrts_fixed) then
        if (phs%p_defined) return
        s_hat = phs%config%sqrts ** 2
      else
        if (.not. phs%p_defined) return
        s_hat = sum (phs%p) ** 2
      end if
      if (sum (phs%m_in)**2 < s_hat .and. sum (phs%m_out)**2 < s_hat) then

```

```

        phs%f(1) = 1 / s_hat * &
        ( lambda (s_hat, phs%m_in (1)**2, phs%m_in (2)**2) &
        * lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2) ) &
        ** 0.25_default
    else
        phs%f(1) = 0
    end if
end select
end subroutine phs_single_compute_factor

```

Evaluation. We uniformly generate  $\cos\theta$  and  $\phi$ , with Jacobian normalized to one.

```

<PHS single: phs single: TBP>+≡
    procedure :: evaluate => phs_single_evaluate

<PHS single: procedures>+≡
    subroutine phs_single_evaluate (phs, c_in, r_in)
    class(phs_single_t), intent(inout) :: phs
    integer, intent(in) :: c_in
    real(default), intent(in), dimension(:) :: r_in
    integer :: n_channel
    real(default), dimension(:), allocatable :: x
    if (phs%p_defined) then
        call phs%select_channel (c_in)
        n_channel = phs%config%n_channel
        allocate (x (phs%config%n_par))
        phs%r(:,c_in) = r_in
        x = r_in
        call compute_kinematics_solid_angle (phs%p, phs%q, x)
        call phs%compute_factor ()
        phs%q_defined = .true.
        phs%r_defined = .true.
    end if
    end subroutine phs_single_evaluate

```

Inverse evaluation.

```

<PHS single: phs single: TBP>+≡
    procedure :: inverse => phs_single_inverse

<PHS single: procedures>+≡
    subroutine phs_single_inverse (phs)
    class(phs_single_t), intent(inout) :: phs
    integer :: n_channel
    real(default), dimension(:), allocatable :: x
    if (phs%p_defined .and. phs%q_defined) then
        call phs%select_channel ()
        n_channel = phs%config%n_channel
        allocate (x (phs%config%n_par))
        call inverse_kinematics_solid_angle (phs%p, phs%q, x)
        phs%r(:,1) = x
        call phs%compute_factor ()
        phs%r_defined = .true.
    end if
    end subroutine phs_single_inverse

```

### 11.3.3 Unit tests

```
<PHS single: public>+≡
  public :: phs_single_test

<PHS single: tests>≡
  subroutine phs_single_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <PHS single: execute tests>
  end subroutine phs_single_test
```

#### Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the azimuthal\_dependence flag.

```
<PHS single: execute tests>≡
  call test (phs_single_1, "phs_single_1", &
    "phase-space configuration", &
    u, results)

<PHS single: tests>+≡
  subroutine phs_single_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data
    real(default) :: sqrts

    write (u, "(A)")  "* Test output: phs_single_1"
    write (u, "(A)")  "* Purpose: initialize and display &
      &phase-space configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)

    write (u, "(A)")  "* Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_single_1"), process_data)

    allocate (phs_single_config_t :: phs_data)
    call phs_data%init (process_data)

    sqrts = 1000._default
    call phs_data%configure (sqrts, azimuthal_dependence=.false.)
```

```

call phs_data%write (u)

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_single_1"

end subroutine phs_single_1

```

## Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

*(PHS single: execute tests)+≡*

```

call test (phs_single_2, "phs_single_2", &
  "phase-space evaluation", &
  u, results)

```

*(PHS single: tests)+≡*

```

subroutine phs_single_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  type(process_constants_t) :: process_data
  real(default) :: sqrts, E
  class(phs_config_t), allocatable, target :: phs_data
  class(phs_t), pointer :: phs => null ()
  type(vector4_t), dimension(2) :: p, q

  write (u, "(A)")  "* Test output: phs_single_2"
  write (u, "(A)")  "* Purpose: test simple two-channel phase space"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)

  write (u, "(A)")  "* Initialize a process and a matching &
    &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_single_2"), process_data)

  allocate (phs_single_config_t :: phs_data)
  call phs_data%init (process_data)

  sqrts = 1000._default
  call phs_data%configure (sqrts)

```

```

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
    &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate (1, [0.5_default, 0.125_default])
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call phs%write (u)

call phs%final ()
deallocate (phs)

call phs_data%final ()

```



```

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "** Test output end: phs_single_2"

end subroutine phs_single_2

```

### Phase space for non-c.m. system

Compute kinematics for given parameters, also invert the calculation. Since this will involve cancellations, we call `pacify` to eliminate numerical noise.

```

<PHS single: execute tests>+≡
  call test (phs_single_3, "phs_single_3", &
    "phase-space evaluation in lab frame", &
    u, results)

<PHS single: tests>+≡
  subroutine phs_single_3 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(process_constants_t) :: process_data
    real(default) :: sqrts, E
    class(phs_config_t), allocatable, target :: phs_data
    class(phs_t), pointer :: phs => null ()
    type(vector4_t), dimension(2) :: p, q
    type(lorentz_transformation_t) :: lt

    write (u, "(A)")  "** Test output: phs_single_3"
    write (u, "(A)")  "** Purpose: test simple two-channel phase space"
    write (u, "(A)")  "**           without c.m. kinematics assumption"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)
    call flavor_init (flv, 25, model)

    write (u, "(A)")  "** Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_single_3"), process_data)

    allocate (phs_single_config_t :: phs_data)
    call phs_data%init (process_data)

    sqrts = 1000._default
    call phs_data%configure (sqrts, cm_frame=.false., sqrts_fixed=.false.)

    call phs_data%write (u)

```

```

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta in lab system"
write (u, "(A)")

lt = boost (0.1_default, 1) * boost (0.3_default, 3)

E = sqrts / 2
p(1) = lt * vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = lt * vector4_moving (E, -sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call vector4_write (p(1), u)
call vector4_write (p(2), u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
    &for x = 0.5, 0.125"
write (u, "(A)")

call phs%set_incoming_momenta (p)
call phs%compute_flux ()

call phs%evaluate (1, [0.5_default, 0.125_default])
call pacify (phs)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract outgoing momenta in lab system"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call vector4_write (q(1), u)
call vector4_write (q(2), u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
call phs_data%configure (sqrts)

```

```

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call pacify (phs)
call phs%write (u)

call phs%final ()
deallocate (phs)

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_single_3"

end subroutine phs_single_3

```

## 11.4 Mappings

Mappings are objects that encode the transformation of the interval  $(0,1)$  to a physical variable  $m^2$  or  $\cos\theta$  (and back), as it is used in the phase space parameterization. The mapping objects contain fixed parameters, the associated methods implement the mapping and inverse mapping operations, including the computation of the Jacobian (phase space factor).

```
< mappings.f90 >≡  
  <File header>  
  
  module mappings  
  
    <Use kinds>  
    use kinds, only: TC !NODEP!  
    <Use strings>  
    use constants, only: pi !NODEP!  
    <Use file utils>  
    use diagnostics !NODEP!  
    use md5  
    use models  
    use flavors  
  
    <Standard module head>  
  
    <Mappings: public>  
  
    <Mappings: parameters>  
  
    <Mappings: types>  
  
    <Mappings: interfaces>  
  
    contains  
  
    <Mappings: procedures>  
  
  end module mappings
```

### 11.4.1 Default parameters

This type holds the default parameters, needed for setting the scale in cases where no mass parameter is available. The contents are public.

```
<Mappings: public>≡  
  public :: mapping_defaults_t  
  
<Mappings: types>≡  
  type :: mapping_defaults_t  
    real(default) :: energy_scale = 10  
    real(default) :: invariant_mass_scale = 10  
    real(default) :: momentum_transfer_scale = 10  
    logical :: step_mapping = .true.  
    logical :: step_mapping_exp = .true.  
    logical :: enable_s_mapping = .false.
```

```

contains
  <Mappings: mapping defaults: TBP>
end type mapping_defaults_t

```

Output.

```

<Mappings: mapping defaults: TBP>≡
  procedure :: write => mapping_defaults_write

<Mappings: procedures>≡
  subroutine mapping_defaults_write (object, unit)
    class(mapping_defaults_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A,ES19.12)") "energy scale = ", &
      object%energy_scale
    write (u, "(3x,A,ES19.12)") "mass scale = ", &
      object%invariant_mass_scale
    write (u, "(3x,A,ES19.12)") "q scale = ", &
      object%momentum_transfer_scale
    write (u, "(3x,A,L1)") "step mapping = ", &
      object%step_mapping
    write (u, "(3x,A,L1)") "step exp. mode = ", &
      object%step_mapping_exp
    write (u, "(3x,A,L1)") "allow s mapping = ", &
      object%enable_s_mapping
  end subroutine mapping_defaults_write

<Mappings: public>+≡
  public :: mapping_defaults_md5sum

<Mappings: procedures>+≡
  function mapping_defaults_md5sum (mapping_defaults) result (md5sum_map)
    character(32) :: md5sum_map
    type(mapping_defaults_t), intent(in) :: mapping_defaults
    integer :: u
    u = free_unit ()
    open (u, status = "scratch")
    write (u, *) mapping_defaults%energy_scale
    write (u, *) mapping_defaults%invariant_mass_scale
    write (u, *) mapping_defaults%momentum_transfer_scale
    write (u, *) mapping_defaults%step_mapping
    write (u, *) mapping_defaults%step_mapping_exp
    write (u, *) mapping_defaults%enable_s_mapping
    rewind (u)
    md5sum_map = md5sum (u)
    close (u)
  end function mapping_defaults_md5sum

```

### 11.4.2 The Mapping type

Each mapping has a type (e.g., s-channel, infrared), a binary code (redundant, but useful for debugging), and a reference particle. The flavor code of this par-

ticle is stored for bookkeeping reasons, what matters are the mass and width of this particle. Furthermore, depending on the type, various mapping parameters can be set and used.

The parameters **a1** to **a3** (for  $m^2$  mappings) and **b1** to **b3** (for  $\cos \theta$  mappings) are values that are stored once to speed up the calculation, if **variable\_limits** is false. The exact meaning of these parameters depends on the mapping type. The limits are fixed if there is a fixed c.m. energy.

```

<Mappings: public>+≡
    public :: mapping_t

<Mappings: types>+≡
    type :: mapping_t
    private
    integer :: type = NO_MAPPING
    integer(TC) :: bincode
    type(flavor_t) :: flv
    real(default) :: mass = 0
    real(default) :: width = 0
    logical :: a_unknown = .true.
    real(default) :: a1 = 0
    real(default) :: a2 = 0
    real(default) :: a3 = 0
    logical :: b_unknown = .true.
    real(default) :: b1 = 0
    real(default) :: b2 = 0
    real(default) :: b3 = 0
    logical :: variable_limits = .true.
end type mapping_t

```

The valid mapping types. The extra type **STEP\_MAPPING** is used only internally.

```

<Mappings: parameters>≡
    <Mapping modes>

```

### 11.4.3 Screen output

Do not write empty mappings.

```

<Mappings: public>+≡
    public :: mapping_write

<Mappings: procedures>+≡
    subroutine mapping_write (map, unit, verbose)
        type(mapping_t), intent(in) :: map
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        character(len=9) :: str
        u = output_unit (unit); if (u < 0) return
        select case(map%type)
        case(S_CHANNEL); str = "s_channel"
        case(COLLINEAR); str = "collinear"
        case(INFRARED); str = "infrared "
        case(RADIATION); str = "radiation"
        case(T_CHANNEL); str = "t_channel"

```

```

case(U_CHANNEL); str = "u_channel"
case(STEP_MAPPING_E); str = "step_exp"
case(STEP_MAPPING_H); str = "step_hyp"
end select
if (map%type /= NO_MAPPING) then
  write (u, '(1x,A,I4,A)') &
    "Branch #", map%bincode, ": " // &
    "Mapping (" // str // ") for particle " // &
    "' ' // char (flavor_get_name (map%flv)) // ' '"
  if (present (verbose)) then
    if (verbose) then
      select case (map%type)
      case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
        write (u, 1) " m/w = ", map%mass, map%width
      case default
        write (u, 1) " m = ", map%mass
      end select
      select case (map%type)
      case (S_CHANNEL, T_CHANNEL, U_CHANNEL, &
        STEP_MAPPING_E, STEP_MAPPING_H, &
        COLLINEAR, INFRARED, RADIATION)
        write (u, 1) " a1/2/3 = ", map%a1, map%a2, map%a3
      end select
      select case (map%type)
      case (T_CHANNEL, U_CHANNEL, COLLINEAR)
        write (u, 1) " b1/2/3 = ", map%b1, map%b2, map%b3
      end select
    end if
  end if
end if
1 format (1x,A,3ES19.12)
end subroutine mapping_write

```

#### 11.4.4 Define a mapping

The initialization routine sets the mapping type and the particle (binary code and flavor code) for which the mapping applies (e.g., a  $Z$  resonance in branch #3). We only need the absolute value of the flavor code.

*(Mappings: public)*+≡

```
public :: mapping_init
```

*(Mappings: procedures)*+≡

```

subroutine mapping_init (mapping, bincode, type, f, model)
  type(mapping_t), intent(inout) :: mapping
  integer(TC), intent(in) :: bincode
  type(string_t), intent(in) :: type
  integer, intent(in), optional :: f
  type(model_t), intent(in), optional, target :: model
  mapping%bincode = bincode
  select case (char (type))
  case ("s_channel"); mapping%type = S_CHANNEL
  case ("collinear"); mapping%type = COLLINEAR
  case ("infrared"); mapping%type = INFRARED

```

```

case ("radiation"); mapping%type = RADIATION
case ("t_channel"); mapping%type = T_CHANNEL
case ("u_channel"); mapping%type = U_CHANNEL
case ("step_exp"); mapping%type = STEP_MAPPING_E
case ("step_hyp"); mapping%type = STEP_MAPPING_H
end select
if (present (f) .and. present (model)) &
    call flavor_init (mapping%flv, abs (f), model)
end subroutine mapping_init

```

This sets the actual mass and width, using a parameter set. Since the auxiliary parameters will only be determined when the mapping is first called, they are marked as unknown.

```

<Mappings: public>+≡
    public :: mapping_set_parameters

<Mappings: procedures>+≡
    subroutine mapping_set_parameters (map, mapping_defaults, variable_limits)
        type(mapping_t), intent(inout) :: map
        type(mapping_defaults_t), intent(in) :: mapping_defaults
        logical, intent(in) :: variable_limits
        if (map%type /= NO_MAPPING) then
            map%mass = flavor_get_mass (map%flv)
            map%width = flavor_get_width (map%flv)
            map%variable_limits = variable_limits
            map%a_unknown = .true.
            map%b_unknown = .true.
            select case (map%type)
            case (S_CHANNEL)
                if (map%mass <= 0) then
                    call mapping_write (map)
                    call msg_fatal &
                        & (" S-channel resonance must have positive mass")
                else if (map%width <= 0) then
                    call mapping_write (map)
                    call msg_fatal &
                        & (" S-channel resonance must have positive width")
                end if
            case (RADIATION)
                map%width = max (map%width, mapping_defaults%energy_scale)
            case (INFRARED, COLLINEAR)
                map%mass = max (map%mass, mapping_defaults%invariant_mass_scale)
            case (T_CHANNEL, U_CHANNEL)
                map%mass = max (map%mass, mapping_defaults%momentum_transfer_scale)
            end select
        end if
    end subroutine mapping_set_parameters

```

For a step mapping the mass and width are set directly, instead of being determined from the flavor parameter (which is meaningless here). They correspond to the effective upper bound of phase space due to a resonance, as opposed to the absolute upper bound.

```

<Mappings: public>+≡

```



```

    public :: mapping_set_step_mapping_parameters
<Mappings: procedures>+≡
    subroutine mapping_set_step_mapping_parameters (map, &
        mass, width, variable_limits)
        type(mapping_t), intent(inout) :: map
        real(default), intent(in) :: mass, width
        logical, intent(in) :: variable_limits
        select case (map%type)
        case (STEP_MAPPING_E, STEP_MAPPING_H)
            map%variable_limits = variable_limits
            map%a_unknown = .true.
            map%b_unknown = .true.
            map%mass = mass
            map%width = width
        end select
    end subroutine mapping_set_step_mapping_parameters

```

#### 11.4.5 Retrieve contents

Return true if there is any / an s-channel mapping.

```

<Mappings: public>+≡
    public :: mapping_is_set
    public :: mapping_is_s_channel

<Mappings: procedures>+≡
    function mapping_is_set (mapping) result (flag)
        logical :: flag
        type(mapping_t), intent(in) :: mapping
        flag = mapping%type /= NO_MAPPING
    end function mapping_is_set

    function mapping_is_s_channel (mapping) result (flag)
        logical :: flag
        type(mapping_t), intent(in) :: mapping
        flag = mapping%type == S_CHANNEL
    end function mapping_is_s_channel

```

Return stored mass and width, respectively.

```

<Mappings: public>+≡
    public :: mapping_get_mass
    public :: mapping_get_width

<Mappings: procedures>+≡
    function mapping_get_mass (mapping) result (mass)
        real(default) :: mass
        type(mapping_t), intent(in) :: mapping
        mass = mapping%mass
    end function mapping_get_mass

    function mapping_get_width (mapping) result (width)
        real(default) :: width
        type(mapping_t), intent(in) :: mapping

```

```

width = mapping%width
end function mapping_get_width

```

#### 11.4.6 Compare mappings

Equality for single mappings and arrays

```

⟨Mappings: public⟩+≡
  public :: operator(==)

⟨Mappings: interfaces⟩≡
  interface operator(==)
    module procedure mapping_equal
  end interface

⟨Mappings: procedures⟩+≡
  function mapping_equal (m1, m2) result (equal)
    type(mapping_t), intent(in) :: m1, m2
    logical :: equal
    if (m1%type == m2%type) then
      select case (m1%type)
        case (NO_MAPPING)
          equal = .true.
        case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
          equal = (m1%mass == m2%mass) .and. (m1%width == m2%width)
        case default
          equal = (m1%mass == m2%mass)
      end select
    else
      equal = .false.
    end if
  end function mapping_equal

```

#### 11.4.7 Mappings of the invariant mass

Inserting an  $x$  value between 0 and 1, we want to compute the corresponding invariant mass  $m^2(x)$  and the jacobian, aka phase space factor  $f(x)$ . We also need the reverse operation.

In general, the phase space factor  $f$  is defined by

$$\frac{1}{s} \int_{m_{\min}^2}^{m_{\max}^2} dm^2 g(m^2) = \int_0^1 dx \frac{1}{s} \frac{dm^2}{dx} g(m^2(x)) = \int_0^1 dx f(x) g(x), \quad (11.5)$$

where thus

$$f(x) = \frac{1}{s} \frac{dm^2}{dx}. \quad (11.6)$$

With this mapping, a function of the form

$$g(m^2) = c \frac{dx(m^2)}{dm^2} \quad (11.7)$$

is mapped to a constant:

$$\frac{1}{s} \int_{m_{\min}^2}^{m_{\max}^2} dm^2 g(m^2) = \int_0^1 dx f(x) g(m^2(x)) = \int_0^1 dx \frac{c}{s}. \quad (11.8)$$

Here is the mapping routine. Input are the available energy squared  $s$ , the limits for  $m^2$ , and the  $x$  value. Output are the  $m^2$  value and the phase space factor  $f$ .

```

<Mappings: public>+≡
  public :: mapping_compute_msq_from_x

<Mappings: procedures>+≡
  subroutine mapping_compute_msq_from_x (map, s, msq_min, msq_max, msq, f, x)
    type(mapping_t), intent(inout) :: map
    real(default), intent(in) :: s, msq_min, msq_max
    real(default), intent(out) :: msq, f
    real(default), intent(in) :: x
    real(default) :: z, msq0, msq1, tmp
    integer :: type
    type = map%type
    if (s == 0) &
      call msg_fatal (" Applying msq mapping for zero energy")
    <Modify mapping type if necessary>
    select case(type)
    case (NO_MAPPING)
      <Constants for trivial msq mapping>
      <Apply trivial msq mapping>
    case (S_CHANNEL)
      <Constants for s-channel resonance mapping>
      <Apply s-channel resonance mapping>
    case (COLLINEAR, INFRARED, RADIATION)
      <Constants for s-channel pole mapping>
      <Apply s-channel pole mapping>
    case (T_CHANNEL, U_CHANNEL)
      <Constants for t-channel pole mapping>
      <Apply t-channel pole mapping>
    case (STEP_MAPPING_E)
      <Constants for exponential step mapping>
      <Apply exponential step mapping>
    case (STEP_MAPPING_H)
      <Constants for hyperbolic step mapping>
      <Apply hyperbolic step mapping>
    case default
      call msg_fatal ( " Attempt to apply undefined msq mapping")
    end select
  end subroutine mapping_compute_msq_from_x

```

The inverse mapping

```

<Mappings: public>+≡
  public :: mapping_compute_x_from_msq

<Mappings: procedures>+≡
  subroutine mapping_compute_x_from_msq (map, s, msq_min, msq_max, msq, f, x)
    type(mapping_t), intent(inout) :: map

```

```

real(default), intent(in) :: s, msq_min, msq_max
real(default), intent(in) :: msq
real(default), intent(out) :: f, x
real(default) :: msq0, msq1, tmp, z
integer :: type
type = map%type
if (s == 0) &
    call msg_fatal (" Applying inverse msq mapping for zero energy")
<Modify mapping type if necessary>
select case (type)
case (NO_MAPPING)
    <Constants for trivial msq mapping>
    <Apply inverse trivial msq mapping>
case (S_CHANNEL)
    <Constants for s-channel resonance mapping>
    <Apply inverse s-channel resonance mapping>
case (COLLINEAR, INFRARED, RADIATION)
    <Constants for s-channel pole mapping>
    <Apply inverse s-channel pole mapping>
case (T_CHANNEL, U_CHANNEL)
    <Constants for t-channel pole mapping>
    <Apply inverse t-channel pole mapping>
case (STEP_MAPPING_E)
    <Constants for exponential step mapping>
    <Apply inverse exponential step mapping>
case (STEP_MAPPING_H)
    <Constants for hyperbolic step mapping>
    <Apply inverse hyperbolic step mapping>
case default
    call msg_fatal ( " Attempt to apply undefined msq mapping")
end select
end subroutine mapping_compute_x_from_msq

```

### Trivial mapping

We simply map the boundaries of the interval  $(m_{\min}, m_{\max})$  to  $(0, 1)$ :

$$m^2 = (1 - x)m_{\min}^2 + xm_{\max}^2; \quad (11.9)$$

the inverse is

$$x = \frac{m^2 - m_{\min}^2}{m_{\max}^2 - m_{\min}^2}. \quad (11.10)$$

Hence

$$f(x) = \frac{m_{\max}^2 - m_{\min}^2}{s}, \quad (11.11)$$

and we have, as required,

$$f(x) \frac{dx}{dm^2} = \frac{1}{s}. \quad (11.12)$$

We store the constant parameters the first time the mapping is called – or, if limits vary, recompute them each time.

*<Constants for trivial msq mapping>*≡

```

if (map%variable_limits .or. map%a_unknown) then
  map%a1 = 0
  map%a2 = msq_max - msq_min
  map%a3 = map%a2 / s
  map%a_unknown = .false.
end if

⟨Apply trivial msq mapping⟩≡
  msq = (1-x) * msq_min + x * msq_max
  f = map%a3

```

```

⟨Apply inverse trivial msq mapping⟩≡
  if (map%a2 /= 0) then
    x = (msq - msq_min) / map%a2
  else
    x = 0
  end if
  f = map%a3

```

Resonance or step mapping does not make much sense if the resonance mass is outside the kinematical bounds. If this is the case, revert to `NO_MAPPING`. This is possible even if the kinematical bounds vary from event to event.

```

⟨Modify mapping type if necessary⟩≡
  select case (type)
  case (S_CHANNEL, STEP_MAPPING_E, STEP_MAPPING_H)
    msq0 = map%mass**2
    if (msq0 < msq_min .or. msq0 > msq_max) type = NO_MAPPING
  end select

```

### Breit-Wigner mapping

A Breit-Wigner resonance with mass  $M$  and width  $\Gamma$  is flattened by the following mapping:

This mapping does not make much sense if the resonance mass is too low. If this is the case, revert to `NO_MAPPING`. There is a tricky point with this if the mass is too high: `msq_max` is not a constant if structure functions are around. However, switching the type depending on the overall energy does not change the integral, it is just another branching point.

$$m^2 = M(M + t\Gamma), \quad (11.13)$$

where

$$t = \tan \left[ (1-x) \arctan \frac{m_{\min}^2 - M^2}{M\Gamma} + x \arctan \frac{m_{\max}^2 - M^2}{M\Gamma} \right]. \quad (11.14)$$

The inverse:

$$x = \frac{\arctan \frac{m^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma}}{\arctan \frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma}} \quad (11.15)$$

The phase-space factor of this transformation is

$$f(x) = \frac{M\Gamma}{s} \left( \arctan \frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma} \right) (1 + t^2). \quad (11.16)$$

This maps any function proportional to

$$g(m^2) = \frac{M\Gamma}{(m^2 - M^2)^2 + M^2\Gamma^2} \quad (11.17)$$

to a constant times  $1/s$ .

```

⟨Constants for s-channel resonance mapping⟩≡
  if (map%variable_limits .or. map%a_unknown) then
    msq0 = map%mass ** 2
    map%a1 = atan ((msq_min - msq0) / (map%mass * map%width))
    map%a2 = atan ((msq_max - msq0) / (map%mass * map%width))
    map%a3 = (map%a2 - map%a1) * (map%mass * map%width) / s
    map%a_unknown = .false.
  end if

⟨Apply s-channel resonance mapping⟩≡
  z = (1-x) * map%a1 + x * map%a2
  if (-pi/2 < z .and. z < pi/2) then
    tmp = tan (z)
    msq = map%mass * (map%mass + map%width * tmp)
    f = map%a3 * (1 + tmp**2)
  else
    msq = 0
    f = 0
  end if

⟨Apply inverse s-channel resonance mapping⟩≡
  tmp = (msq - msq0) / (map%mass * map%width)
  x = (atan (tmp) - map%a1) / (map%a2 - map%a1)
  f = map%a3 * (1 + tmp**2)

```

### Mapping for massless splittings

This mapping accounts for approximately scale-invariant behavior where  $\ln M^2$  is evenly distributed.

$$m^2 = m_{\min}^2 + M^2 (\exp(xL) - 1) \quad (11.18)$$

where

$$L = \ln \left( \frac{m_{\max}^2 - m_{\min}^2}{M^2} + 1 \right). \quad (11.19)$$

The inverse:

$$x = \frac{1}{L} \ln \left( \frac{m^2 - m_{\min}^2}{M^2} + 1 \right) \quad (11.20)$$

The constant  $M$  is a characteristic scale. Above this scale ( $m^2 - m_{\min}^2 \gg M^2$ ), this mapping behaves like  $x \propto \ln m^2$ , while below the scale it reverts to a linear mapping.

The phase-space factor is

$$f(x) = \frac{M^2}{s} \exp(xL) L. \quad (11.21)$$

A function proportional to

$$g(m^2) = \frac{1}{(m^2 - m_{\min}^2) + M^2} \quad (11.22)$$

is mapped to a constant, i.e., a simple pole near  $m_{\min}$  with a regulator mass  $M$ .

This type of mapping is useful for massless collinear and infrared singularities, where the scale is stored as the mass parameter. In the radiation case (IR radiation off massive particle), the heavy particle width is the characteristic scale.

```

⟨Constants for s-channel pole mapping⟩≡
  if (map%variable_limits .or. map%a_unknown) then
    if (type == RADIATION) then
      msq0 = map%width**2
    else
      msq0 = map%mass**2
    end if
    map%a1 = msq0
    map%a2 = log ((msq_max - msq_min) / msq0 + 1)
    map%a3 = map%a2 / s
    map%a_unknown = .false.
  end if

⟨Apply s-channel pole mapping⟩≡
  msq1 = map%a1 * exp (x * map%a2)
  msq = msq1 - map%a1 + msq_min
  f = map%a3 * msq1

⟨Apply inverse s-channel pole mapping⟩≡
  msq1 = msq - msq_min + map%a1
  x = log (msq1 / map%a1) / map%a2
  f = map%a3 * msq1

```

### Mapping for t-channel poles

This is also approximately scale-invariant, and we use the same type of mapping as before. However, we map  $1/x$  singularities at both ends of the interval; again, the mapping becomes linear when the distance is less than  $M^2$ :

$$m^2 = \begin{cases} m_{\min}^2 + M^2 (\exp(xL) - 1) & \text{for } 0 < x < \frac{1}{2} \\ m_{\max}^2 - M^2 (\exp((1-x)L) - 1) & \text{for } \frac{1}{2} \leq x < 1 \end{cases} \quad (11.23)$$

where

$$L = 2 \ln \left( \frac{m_{\max}^2 - m_{\min}^2}{2M^2} + 1 \right). \quad (11.24)$$

The inverse:

$$x = \begin{cases} \frac{1}{L} \ln \left( \frac{m^2 - m_{\min}^2}{M^2} + 1 \right) & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1 - \frac{1}{L} \ln \left( \frac{m_{\max}^2 - m^2}{M^2} + 1 \right) & \text{for } m^2 \geq (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \quad (11.25)$$

The phase-space factor is

$$f(x) = \begin{cases} \frac{M^2}{s} \exp(xL) L. & \text{for } 0 < x < \frac{1}{2} \\ \frac{M^2}{s} \exp((1-x)L) L. & \text{for } \frac{1}{2} \leq x < 1 \end{cases} \quad (11.26)$$

A (continuous) function proportional to

$$g(m^2) = \begin{cases} 1/(m^2 - m_{\min}^2) + M^2 & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1/((m_{\max}^2 - m^2) + M^2) & \text{for } m^2 \geq (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \quad (11.27)$$

is mapped to a constant by this mapping, i.e., poles near both ends of the interval.

```

⟨Constants for t-channel pole mapping⟩≡
  if (map%variable_limits .or. map%a_unknown) then
    msq0 = map%mass**2
    map%a1 = msq0
    map%a2 = 2 * log ((msq_max - msq_min)/(2*msq0) + 1)
    map%a3 = map%a2 / s
    map%a_unknown = .false.
  end if

⟨Apply t-channel pole mapping⟩≡
  if (x < .5_default) then
    msq1 = map%a1 * exp (x * map%a2)
    msq = msq1 - map%a1 + msq_min
  else
    msq1 = map%a1 * exp ((1-x) * map%a2)
    msq = -(msq1 - map%a1) + msq_max
  end if
  f = map%a3 * msq1

⟨Apply inverse t-channel pole mapping⟩≡
  if (msq < (msq_max + msq_min)/2) then
    msq1 = msq - msq_min + map%a1
    x = log (msq1/map%a1) / map%a2
  else
    msq1 = msq_max - msq + map%a1
    x = 1 - log (msq1/map%a1) / map%a2
  end if
  f = map%a3 * msq1

```

#### 11.4.8 Step mapping

Step mapping is useful when the allowed range for a squared-mass variable is large, but only a fraction at the lower end is populated because the particle in question is an (off-shell) decay product of a narrow resonance. I.e., if the resonance was forced to be on-shell, the upper end of the range would be the resonance mass, minus the effective (real or resonance) mass of the particle(s) in the sibling branch of the decay.

The edge of this phase space section has a width which is determined by the width of the parent, plus the width of the sibling branch. (The widths might be added in quadrature, but this precision is probably not important.)

##### Fermi function

A possible mapping is derived from the Fermi function which has precisely this behavior. The Fermi function is given by

$$f(x) = \frac{1}{1 + \exp \frac{x-\mu}{\gamma}} \quad (11.28)$$

where  $x$  is taken as the invariant mass squared,  $\mu$  is the invariant mass squared of the edge, and  $\gamma$  is the effective width which is given by the widths of the



parent and the sibling branch. (Widths might be added in quadrature, but we do not require this level of precision.)

$$x = \frac{m^2 - m_{\min}^2}{\Delta m^2} \quad (11.29)$$

$$\mu = \frac{m_{\max, \text{eff}}^2 - m_{\min}^2}{\Delta m^2} \quad (11.30)$$

$$\gamma = \frac{2m_{\max, \text{eff}}\Gamma}{\Delta m^2} \quad (11.31)$$

with

$$\Delta m^2 = m_{\max}^2 - m_{\min}^2 \quad (11.32)$$

$m^2$  is thus given by

$$m^2(x) = xm_{\max}^2 + (1-x)m_{\min}^2 \quad (11.33)$$

For the mapping, we compute the integral  $g(x)$  of the Fermi function, normalized such that  $g(0) = 0$  and  $g(1) = 1$ . We introduce the abbreviations

$$\alpha = 1 - \gamma \ln \frac{1 + \beta e^{1/\gamma}}{1 + \beta} \quad (11.34)$$

$$\beta = e^{-\mu/\gamma} \quad (11.35)$$

and obtain

$$g(x) = \frac{1}{\alpha} \left( x - \gamma \ln \frac{1 + \beta e^{x/\gamma}}{1 + \beta} \right) \quad (11.36)$$

The actual mapping is the inverse function  $h(y) = g^{-1}(y)$ ,

$$h(y) = -\gamma \ln \left( e^{-\alpha y/\gamma} (1 + \beta) - \beta \right) \quad (11.37)$$

The Jacobian is

$$\frac{dh}{dy} = \alpha \left( 1 - e^{\alpha y/\gamma} \frac{\beta}{1 + \beta} \right)^{-1} \quad (11.38)$$

which is equal to  $1/(dg/dx)$ , namely

$$\frac{dg}{dx} = \frac{1}{\alpha} \frac{1}{1 + \beta e^{x/\gamma}} \quad (11.39)$$

The final result is

$$\int_{m_{\min}^2}^{m_{\max}^2} dm^2 F(m^2) = \Delta m^2 \int_0^1 dx F(m^2(x)) \quad (11.40)$$

$$= \Delta m^2 \int_0^1 dy F(m^2(h(y))) \frac{dh}{dy} \quad (11.41)$$

Here is the implementation. We fill **a1**, **a2**, **a3** with  $\alpha, \beta, \gamma$ , respectively.

```

⟨Constants for exponential step mapping⟩≡
  if (map%variable_limits .or. map%a_unknown) then
    map%a3 = max (2 * map%mass * map%width / (msq_max - msq_min), 0.01_default)
    map%a2 = exp (- (map%mass**2 - msq_min) / (msq_max - msq_min) &
                  / map%a3)
    map%a1 = 1 - map%a3 * log ((1 + map%a2 * exp (1 / map%a3)) / (1 + map%a2))
  end if

```

```

⟨Apply exponential step mapping⟩≡
  tmp = exp (- x * map%a1 / map%a3) * (1 + map%a2)
  z = - map%a3 * log (tmp - map%a2)
  msq = z * msq_max + (1 - z) * msq_min
  f = map%a1 / (1 - map%a2 / tmp) * (msq_max - msq_min) / s

⟨Apply inverse exponential step mapping⟩≡
  z = (msq - msq_min) / (msq_max - msq_min)
  tmp = 1 + map%a2 * exp (z / map%a3)
  x = (z - map%a3 * log (tmp / (1 + map%a2))) &
    / map%a1
  f = map%a1 * tmp * (msq_max - msq_min) / s

```

## Hyperbolic mapping

The Fermi function has the drawback that it decreases exponentially. It might be preferable to take a function with a power-law decrease, such that the high-mass region is not completely depopulated.

Here, we start with the actual mapping which we take as

$$h(y) = \frac{b}{a-y} - \frac{b}{a} + \mu y \quad (11.42)$$

with the abbreviation

$$a = \frac{1}{2} \left( 1 + \sqrt{1 + \frac{4b}{1-\mu}} \right) \quad (11.43)$$

This is a hyperbola in the  $xy$  plane. The derivative is

$$\frac{dh}{dy} = \frac{b}{(a-y)^2} + \mu \quad (11.44)$$

The constants correspond to

$$\mu = \frac{m_{\text{max,eff}}^2 - m_{\text{min}}^2}{\Delta m^2} \quad (11.45)$$

$$b = \frac{1}{\mu} \left( \frac{2m_{\text{max,eff}}\Gamma}{\Delta m^2} \right)^2 \quad (11.46)$$

The inverse function is the solution of a quadratic equation,

$$g(x) = \frac{1}{2} \left[ \left( a + \frac{x}{\mu} + \frac{b}{a\mu} \right) - \sqrt{\left( a - \frac{x}{\mu} \right)^2 + 2\frac{b}{a\mu} \left( a + \frac{x}{\mu} \right) + \left( \frac{b}{a\mu} \right)^2} \right] \quad (11.47)$$

The constants  $a_{1,2,3}$  are identified with  $a, b, \mu$ .

```

⟨Constants for hyperbolic step mapping⟩≡
  if (map%variable_limits .or. map%a_unknown) then
    map%a3 = (map%mass**2 - msq_min) / (msq_max - msq_min)
    map%a2 = max ((2 * map%mass * map%width / (msq_max - msq_min))**2 &
      / map%a3, 1e-6_default)
    map%a1 = (1 + sqrt (1 + 4 * map%a2 / (1 - map%a3))) / 2
  end if

```

```

⟨Apply hyperbolic step mapping⟩≡
  z = map%a2 / (map%a1 - x) - map%a2 / map%a1 + map%a3 * x
  msq = z * msq_max + (1 - z) * msq_min
  f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s

⟨Apply inverse hyperbolic step mapping⟩≡
  z = (msq - msq_min) / (msq_max - msq_min)
  tmp = map%a2 / (map%a1 * map%a3)
  x = ((map%a1 + z / map%a3 + tmp) &
        - sqrt((map%a1 - z / map%a3)**2 + 2 * tmp * (map%a1 + z / map%a3) &
              + tmp**2)) / 2
  f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s

```

### 11.4.9 Mappings of the polar angle

The other type of singularity, a simple pole just outside the integration region, can occur in the integration over  $\cos\theta$ . This applies to exchange of massless (or light) particles.

Double poles (Coulomb scattering) are also possible, but only in certain cases. These are also handled by the single-pole mapping.

The mapping is analogous to the previous  $m^2$  pole mapping, but with a different normalization and notation of variables:

$$\frac{1}{2} \int_{-1}^1 d\cos\theta g(\theta) = \int_0^1 dx \frac{d\cos\theta}{dx} g(\theta(x)) = \int_0^1 dx f(x) g(x), \quad (11.48)$$

where thus

$$f(x) = \frac{1}{2} \frac{d\cos\theta}{dx}. \quad (11.49)$$

With this mapping, a function of the form

$$g(\theta) = c \frac{dx(\cos\theta)}{d\cos\theta} \quad (11.50)$$

is mapped to a constant:

$$\int_{-1}^1 d\cos\theta g(\theta) = \int_0^1 dx f(x) g(\theta(x)) = \int_0^1 dx c. \quad (11.51)$$

```

⟨Mappings: public⟩+≡
  public :: mapping_compute_ct_from_x

⟨Mappings: procedures⟩+≡
  subroutine mapping_compute_ct_from_x (map, s, ct, st, f, x)
    type(mapping_t), intent(inout) :: map
    real(default), intent(in) :: s
    real(default), intent(out) :: ct, st, f
    real(default), intent(in) :: x
    real(default) :: tmp, ct1
    select case (map%type)
    case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
          STEP_MAPPING_E, STEP_MAPPING_H)
      ⟨Apply trivial ct mapping⟩
    case (T_CHANNEL, U_CHANNEL, COLLINEAR)

```

```

    <Constants for ct pole mapping>
    <Apply ct pole mapping>
    case default
        call msg_fatal (" Attempt to apply undefined ct mapping")
    end select
end subroutine mapping_compute_ct_from_x

<Mappings: public>+≡
    public :: mapping_compute_x_from_ct

<Mappings: procedures>+≡
    subroutine mapping_compute_x_from_ct (map, s, ct, f, x)
        type(mapping_t), intent(inout) :: map
        real(default), intent(in) :: s
        real(default), intent(in) :: ct
        real(default), intent(out) :: f, x
        real(default) :: ct1
        select case (map%type)
            case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
                STEP_MAPPING_E, STEP_MAPPING_H)
                <Apply inverse trivial ct mapping>
            case (T_CHANNEL, U_CHANNEL, COLLINEAR)
                <Constants for ct pole mapping>
                <Apply inverse ct pole mapping>
            case default
                call msg_fatal (" Attempt to apply undefined inverse ct mapping")
            end select
        end select
    end subroutine mapping_compute_x_from_ct

```

## Trivial mapping

This is just the mapping of the interval  $(-1, 1)$  to  $(0, 1)$ :

$$\cos \theta = -1 + 2x \quad (11.52)$$

and

$$f(x) = 1 \quad (11.53)$$

with the inverse

$$x = \frac{1 + \cos \theta}{2} \quad (11.54)$$

```

<Apply trivial ct mapping>≡
    tmp = 2 * (1-x)
    ct = 1 - tmp
    st = sqrt (tmp * (2-tmp))
    f = 1

<Apply inverse trivial ct mapping>≡
    x = (ct + 1) / 2
    f = 1

```

### Pole mapping

As above for  $m^2$ , we simultaneously map poles at both ends of the  $\cos \theta$  interval. The formulae are completely analogous:

$$\cos \theta = \begin{cases} \frac{M^2}{s} [\exp(xL) - 1] - 1 & \text{for } x < \frac{1}{2} \\ -\frac{M^2}{s} [\exp((1-x)L) - 1] + 1 & \text{for } x \geq \frac{1}{2} \end{cases} \quad (11.55)$$

where

$$L = 2 \ln \frac{M^2 + s}{M^2}. \quad (11.56)$$

Inverse:

$$x = \begin{cases} \frac{1}{2L} \ln \frac{1 + \cos \theta + M^2/s}{M^2/s} & \text{for } \cos \theta < 0 \\ 1 - \frac{1}{2L} \ln \frac{1 - \cos \theta + M^2/s}{M^2/s} & \text{for } \cos \theta \geq 0 \end{cases} \quad (11.57)$$

The phase-space factor:

$$f(x) = \begin{cases} \frac{M^2}{s} \exp(xL) L & \text{for } x < \frac{1}{2} \\ \frac{M^2}{s} \exp((1-x)L) L & \text{for } x \geq \frac{1}{2} \end{cases} \quad (11.58)$$

```

<Constants for ct pole mapping>≡
  if (map%variable_limits .or. map%b_unknown) then
    map%b1 = map%mass**2 / s
    map%b2 = log ((map%b1 + 1) / map%b1)
    map%b3 = 0
    map%b_unknown = .false.
  end if

<Apply ct pole mapping>≡
  if (x < .5_default) then
    ct1 = map%b1 * exp (2 * x * map%b2)
    ct = ct1 - map%b1 - 1
  else
    ct1 = map%b1 * exp (2 * (1-x) * map%b2)
    ct = -(ct1 - map%b1) + 1
  end if
  if (ct >= -1 .and. ct <= 1) then
    st = sqrt (1 - ct**2)
    f = ct1 * map%b2
  else
    ct = 1; st = 0; f = 0
  end if

<Apply inverse ct pole mapping>≡
  if (ct < 0) then
    ct1 = ct + map%b1 + 1
    x = log (ct1 / map%b1) / (2 * map%b2)
  else
    ct1 = -ct + map%b1 + 1
    x = 1 - log (ct1 / map%b1) / (2 * map%b2)
  end if
  f = ct1 * map%b2

```

## 11.5 Phase-space trees

The phase space evaluation is organized in terms of trees, where each branch corresponds to three integrations:  $m^2$ ,  $\cos\theta$ , and  $\phi$ . The complete tree thus makes up a specific parameterization of the multidimensional phase-space integral. For the multi-channel integration, the phase-space tree is a single channel.

The trees imply mappings of formal Feynman tree graphs into arrays of integer numbers: Each branch, corresponding to a particular line in the graph, is assigned an integer code  $c$  (with kind value  $\text{TC} = \text{tree code}$ ).

In this integer, each bit determines whether a particular external momentum flows through the line. The external branches therefore have codes 1, 2, 4, 8, ... An internal branch has those bits ORed corresponding to the momenta flowing through it. For example, a branch with momentum  $p_1 + p_4$  has code  $2^0 + 2^3 = 1 + 8 = 9$ .

There is a two-fold ambiguity: Momentum conservation implies that the branch with code

$$c_0 = \sum_{i=1}^{n(\text{ext})} 2^{i-1} \quad (11.59)$$

i.e. the branch with momentum  $p_1 + p_2 + \dots p_n$  has momentum zero, which is equivalent to tree code 0 by definition. Correspondingly,

$$c \quad \text{and} \quad c_0 - c = c \text{ XOR } c_0 \quad (11.60)$$

are equivalent. E.g., if there are five externals with codes  $c = 1, 2, 4, 8, 16$ , then  $c = 9$  and  $\bar{c} = 31 - 9 = 22$  are equivalent.

This ambiguity may be used to assign a direction to the line: If all momenta are understood as outgoing,  $c = 9$  in the example above means  $p_1 + p_4$ , but  $c = 22$  means  $p_2 + p_3 + p_5 = -(p_1 + p_4)$ .

Here we make use of the ambiguity in a slightly different way. First, the initial particles are singled out as those externals with the highest bits, the IN-bits. (Here: 8 and 16 for a  $2 \rightarrow 3$  scattering process, 16 only for a  $1 \rightarrow 4$  decay.) Then we invert those codes where all IN-bits are set. For a decay process this maps each tree of an equivalence class onto a unique representative (that one with the smallest integer codes). For a scattering process we proceed further:

The ambiguity remains in all branches where only one IN-bit is set, including the initial particles. If there are only externals with this property, we have an  $s$ -channel graph which we leave as it is. In all other cases, an internal with only one IN-bit is a  $t$ -channel line, which for phase space integration should be associated with one of the initial momenta as a reference axis. We take that one whose bit is set in the current tree code. (E.g., for branch  $c = 9$  we use the initial particle  $c = 8$  as reference axis, whereas for the same branch we would take  $c = 16$  if it had been assigned  $\bar{c} = 31 - 9 = 22$  as tree code.) Thus, different ways of coding the same  $t$ -channel graph imply different phase space parameterizations.

$s$ -channel graphs have a unique parameterization. The same sets of parameterizations are used for  $t$ -channel graphs, except for the reference frames of their angular parts. We map each  $t$ -channel graph onto an  $s$ -channel graph as follows:

Working in ascending order, for each  $t$ -channel line (whose code has exactly one IN-bit set) the attached initial line is flipped upstream, while the outgoing

line is flipped downstream. (This works only if  $t$ -channel graphs are always parameterized beginning at their outer vertices, which we require as a restriction.) After all possible flips have been applied, we have an  $s$ -channel graph. We only have to remember the initial particle a vertex was originally attached to.

```

<phs_trees.f90>≡
  <File header>

  module phs_trees

    <Use kinds>
      use kinds, only: TC !NODEP!
    <Use strings>
      use constants, only: twopi, twopi2, twopi5 !NODEP!
    <Use file utils>
      use diagnostics !NODEP!
      use lorentz !NODEP!
      use permutations, only: permutation_t, permutation_size
      use permutations, only: permutation_init, permutation_find
      use permutations, only: tc_decay_level, tc_permute
      use models
      use flavors
      use mappings

    <Standard module head>

    <PHS trees: public>

    <PHS trees: types>

    contains

    <PHS trees: procedures>

  end module phs_trees

```

### 11.5.1 Particles

We define a particle type which contains only four-momentum and invariant mass squared, and a flag that tells whether the momentum is filled or not.

```

<PHS trees: public>≡
  public :: phs_prt_t

<PHS trees: types>≡
  type :: phs_prt_t
    private
    logical :: defined = .false.
    type(vector4_t) :: p
    real(default) :: p2
  end type phs_prt_t

```

Set contents:

```

<PHS trees: public>+≡
  public :: phs_prt_set_defined

```

```

public :: phs_prt_set_undefined
public :: phs_prt_set_momentum
public :: phs_prt_set_msq

<PHS trees: procedures>≡
elemental subroutine phs_prt_set_defined (prt)
  type(phs_prt_t), intent(inout) :: prt
  prt%defined = .true.
end subroutine phs_prt_set_defined

elemental subroutine phs_prt_set_undefined (prt)
  type(phs_prt_t), intent(inout) :: prt
  prt%defined = .false.
end subroutine phs_prt_set_undefined

elemental subroutine phs_prt_set_momentum (prt, p)
  type(phs_prt_t), intent(inout) :: prt
  type(vector4_t), intent(in) :: p
  prt%p = p
end subroutine phs_prt_set_momentum

elemental subroutine phs_prt_set_msq (prt, p2)
  type(phs_prt_t), intent(inout) :: prt
  real(default), intent(in) :: p2
  prt%p2 = p2
end subroutine phs_prt_set_msq

```

Access methods:

```

<PHS trees: public>+≡
public :: phs_prt_is_defined
public :: phs_prt_get_momentum
public :: phs_prt_get_msq

<PHS trees: procedures>+≡
elemental function phs_prt_is_defined (prt) result (defined)
  logical :: defined
  type(phs_prt_t), intent(in) :: prt
  defined = prt%defined
end function phs_prt_is_defined

elemental function phs_prt_get_momentum (prt) result (p)
  type(vector4_t) :: p
  type(phs_prt_t), intent(in) :: prt
  p = prt%p
end function phs_prt_get_momentum

elemental function phs_prt_get_msq (prt) result (p2)
  real(default) :: p2
  type(phs_prt_t), intent(in) :: prt
  p2 = prt%p2
end function phs_prt_get_msq

```

Addition of momenta (invariant mass square is computed).

```

<PHS trees: public>+≡

```



```

public :: phs_prt_combine
<PHS trees: procedures>+≡
elemental subroutine phs_prt_combine (prt, prt1, prt2)
  type(phs_prt_t), intent(inout) :: prt
  type(phs_prt_t), intent(in) :: prt1, prt2
  prt%defined = .true.
  prt%p = prt1%p + prt2%p
  prt%p2 = prt%p ** 2
end subroutine phs_prt_combine

```

Output

```

<PHS trees: public>+≡
public :: phs_prt_write
<PHS trees: procedures>+≡
subroutine phs_prt_write (prt, unit)
  type(phs_prt_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit); if (u < 0) return
  if (prt%defined) then
    call vector4_write (prt%p, u)
    write (u, "(1x,A,1x,ES19.12)") "T = ", prt%p2
  else
    write (u, "(3x,A)") "[undefined]"
  end if
end subroutine phs_prt_write

```

## 11.5.2 The phase-space tree type

### Definition

In the concrete implementation, each branch  $c$  may have two *daughters*  $c_1$  and  $c_2$  such that  $c_1 + c_2 = c$ , a *sibling*  $c_s$  and a *mother*  $c_m$  such that  $c + c_s = c_m$ , and a *friend* which is kept during flips, such that it can indicate a fixed reference frame. Absent entries are set  $c = 0$ .

First, declare the branch type. There is some need to have this public. Give initializations for all components, so no `init` routine is necessary. The branch has some information about the associated coordinates and about connections.

```

<PHS trees: types>+≡
type :: phs_branch_t
  private
  logical :: set = .false.
  logical :: inverted_decay = .false.
  logical :: inverted_axis = .false.
  integer(TC) :: mother = 0
  integer(TC) :: sibling = 0
  integer(TC) :: friend = 0
  integer(TC) :: origin = 0
  integer(TC), dimension(2) :: daughter = 0
  integer :: firstborn = 0

```

```

        logical :: has_children = .false.
        logical :: has_friend = .false.
    end type phs_branch_t

```

The tree type: No initialization, this is done by `phs_tree_init`. In addition to the branch array which

The branches are collected in an array which holds all possible branches, of which only a few are set. After flips have been applied, the branch  $c_M = \sum_{i=1}^{n(\text{fin})} 2^{i-1}$  must be there, indicating the mother of all decay products. In addition, we should check for consistency at the beginning.

`n_branches` is the number of those actually set. `n externals` defines the number of significant bit, and `mask` is a code where all bits are set. Analogous: `n_in` and `mask_in` for the incoming particles.

The mapping array contains the mappings associated to the branches (corresponding indices). The array `mass_sum` contains the sum of the real masses of the external final-state particles associated to the branch. During phase-space evaluation, this determines the boundaries.

```

<PHS trees: public>+≡
    public :: phs_tree_t

<PHS trees: types>+≡
    type :: phs_tree_t
        private
        integer :: n_branches, n externals, n_in, n_msq, n_angles
        integer(TC) :: n_branches_tot, n_branches_out
        integer(TC) :: mask, mask_in, mask_out
        type(phs_branch_t), dimension(:), allocatable :: branch
        type(mapping_t), dimension(:), allocatable :: mapping
        real(default), dimension(:), allocatable :: mass_sum
        real(default), dimension(:), allocatable :: effective_mass
        real(default), dimension(:), allocatable :: effective_width
    end type phs_tree_t

```

The maximum number of external particles that can be represented is related to the bit size of the integer that stores binary codes. With the default integer of 32 bit on common machines, this is more than enough space. If TC is actually the default integer kind, there is no need to keep it separate, but doing so marks this as a special type of integer. So, just state that the maximum number is 32:

```

<Limits: public parameters>+≡
    integer, parameter, public :: MAX_EXTERNAL = 32

```

### Constructor and destructor

Allocate memory for a phase-space tree with given number of externals and incoming. The number of allocated branches can easily become large, but appears manageable for realistic cases, e.g., for `n_in=2` and `n_out=8` we get  $2^{10} - 1 = 1023$ .

```

<PHS trees: public>+≡
    public :: phs_tree_init
    public :: phs_tree_final

```

Here we set the masks for incoming and for all externals.

```

<PHS trees: procedures>+≡
  elemental subroutine phs_tree_init (tree, n_in, n_out, n_masses, n_angles)
    type(phs_tree_t), intent(inout) :: tree
    integer, intent(in) :: n_in, n_out, n_masses, n_angles
    integer(TC) :: i
    tree%n_externals = n_in + n_out
    tree%n_branches_tot = 2**(n_in+n_out) - 1
    tree%n_branches_out = 2**n_out - 1
    tree%mask = 0
    do i = 0, n_in + n_out - 1
      tree%mask = ibset (tree%mask, i)
    end do
    tree%n_in = n_in
    tree%mask_in = 0
    do i = n_out, n_in + n_out - 1
      tree%mask_in = ibset (tree%mask_in, i)
    end do
    tree%mask_out = ieor (tree%mask, tree%mask_in)
    tree%n_msq = n_masses
    tree%n_angles = n_angles
    allocate (tree%branch (tree%n_branches_tot))
    tree%n_branches = 0
    allocate (tree%mapping (tree%n_branches_out))
    allocate (tree%mass_sum (tree%n_branches_out))
    allocate (tree%effective_mass (tree%n_branches_out))
    allocate (tree%effective_width (tree%n_branches_out))
  end subroutine phs_tree_init

  elemental subroutine phs_tree_final (tree)
    type(phs_tree_t), intent(inout) :: tree
    deallocate (tree%branch)
    deallocate (tree%mapping)
    deallocate (tree%mass_sum)
    deallocate (tree%effective_mass)
    deallocate (tree%effective_width)
  end subroutine phs_tree_final

```

## Screen output

Write only the branches that are set:

```

<PHS trees: public>+≡
  public :: phs_tree_write

<PHS trees: procedures>+≡
  subroutine phs_tree_write (tree, unit)
    type(phs_tree_t), intent(in) :: tree
    integer, intent(in), optional :: unit
    integer :: u
    integer(TC) :: k
    u = output_unit (unit); if (u < 0) return
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
      'External:', tree%n_externals, 'Mask:', tree%mask
  end subroutine phs_tree_write

```

```

write (u, '(3X,A,1x,I0,5X,A,I3)') &
    'Incoming:', tree%n_in, 'Mask:', tree%mask_in
write (u, '(3X,A,1x,I0,5X,A,I3)') &
    'Branches:', tree%n_branches
do k = size (tree%branch), 1, -1
    if (tree%branch(k)%set) &
        call phs_branch_write (tree%branch(k), unit=unit, kval=k)
end do
do k = 1, size (tree%mapping)
    call mapping_write (tree%mapping (k), unit, verbose=.true.)
end do
write (u, "(3x,A)") "Arrays: mass_sum, effective_mass, effective_width"
do k = 1, size (tree%mass_sum)
    if (tree%branch(k)%set) then
        write (u, "(5x,I0,3(2x,ES19.12))") k, tree%mass_sum(k), &
            tree%effective_mass(k), tree%effective_width(k)
    end if
end do
end subroutine phs_tree_write

subroutine phs_branch_write (b, unit, kval)
    type(phs_branch_t), intent(in) :: b
    integer, intent(in), optional :: unit
    integer(TC), intent(in), optional :: kval
    integer :: u
    integer(TC) :: k
    character(len=6) :: tmp
    character(len=1) :: firstborn(2), sign_decay, sign_axis
    integer :: i
    u = output_unit (unit); if (u < 0) return
    k = 0; if (present (kval)) k = kval
    if (b%origin /= 0) then
        write(tmp, '(A,I4,A)') '(', b%origin, ')'
    else
        tmp = ' '
    end if
    do i=1, 2
        if (b%firstborn == i) then
            firstborn(i) = "*"
        else
            firstborn(i) = " "
        end if
    end do
    if (b%inverted_decay) then
        sign_decay = "-"
    else
        sign_decay = "+"
    end if
    if (b%inverted_axis) then
        sign_axis = "-"
    else
        sign_axis = "+"
    end if
    if (b%has_children) then

```

```

        if (b%has_friend) then
            write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A,1x,I0)') &
                & ' ', k, tmp, &
                & 'Daughters: ', &
                & b%daughter(1), firstborn(1), &
                & b%daughter(2), firstborn(2), sign_decay, &
                & 'Friend: ', b%friend
        else
            write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A)') &
                & ' ', k, tmp, &
                & 'Daughters: ', &
                & b%daughter(1), firstborn(1), &
                & b%daughter(2), firstborn(2), sign_decay, &
                & '(axis '//sign_axis//')'
        end if
    else
        write(u,'(5X,I0)') k
    end if
end subroutine phs_branch_write

```

### 11.5.3 PHS tree setup

#### Transformation into an array of branch codes and back

Assume that the tree/array has been created before with the appropriate length and is empty.

```

<PHS trees: public>+≡
    public :: phs_tree_from_array

<PHS trees: procedures>+≡
    subroutine phs_tree_from_array (tree, a)
        type(phs_tree_t), intent(inout) :: tree
        integer(TC), dimension(:), intent(in) :: a
        integer :: i
        integer(TC) :: k
        <Set branches from array a>
        <Set external branches if necessary>
        <Check number of branches>
        <Determine the connections>
    contains
        <Subroutine: set relatives>
    end subroutine phs_tree_from_array

```

First, set all branches specified by the user. If all IN-bits are set, we invert the branch code.

```

<Set branches from array a>≡
    do i=1, size(a)
        k = a(i)
        if (iand(k, tree%mask_in) == tree%mask_in) k = ieor(tree%mask, k)
        tree%branch(k)%set = .true.
        tree%n_branches = tree%n_branches+1
    end do

```

The external branches are understood, so set them now if not yet done. In all cases ensure that the representative with one bit set is used, except for decays where the in-particle is represented by all OUT-bits set instead.

```

<Set external branches if necessary>≡
do i=0, tree%n_externals-1
  k = ibset(0,i)
  if (iand(k, tree%mask_in) == tree%mask_in) k = ieor(tree%mask, k)
  if (tree%branch(ieor(tree%mask, k))%set) then
    tree%branch(ieor(tree%mask, k))%set = .false.
    tree%branch(k)%set = .true.
  else if (.not.tree%branch(k)%set) then
    tree%branch(k)%set = .true.
    tree%n_branches = tree%n_branches+1
  end if
end do

```

Now the number of branches set can be checked. Here we assume that the tree is binary. For three externals there are three branches in total, and for each additional external branch we get another internal one.

```

<Check number of branches>≡
if (tree%n_branches /= tree%n_externals*2-3) then
  call phs_tree_write (tree)
  call msg_bug &
    & (" Wrong number of branches set in phase space tree")
end if

```

For all branches that are set, except for the externals, we try to find the daughter branches:

```

<Determine the connections>≡
do k=1, size (tree%branch)
  if (tree%branch(k)%set .and. tc_decay_level (k) /= 1) then
    call branch_set_relatives(k)
  end if
end do

```

To this end, we scan all codes less than the current code, whether we can find two branches which are set and which together give the current code. After that, the tree may still not be connected, but at least we know if a branch does not have daughters: This indicates some inconsistency.

The algorithm ensures that, at this stage, the first daughter has a smaller code value than the second one.

```

<Subroutine: set relatives>≡
subroutine branch_set_relatives (k)
  integer(TC), intent(in) :: k
  integer(TC) :: m,n
  do m=1, k-1
    if(iand(k,m)==m) then
      n = ieor(k,m)
      if ( tree%branch(m)%set .and. tree%branch(n)%set ) then
        tree%branch(k)%daughter(1) = m; tree%branch(k)%daughter(2) = n
        tree%branch(m)%mother      = k; tree%branch(n)%mother      = k
        tree%branch(m)%sibling     = n; tree%branch(n)%sibling     = m
        tree%branch(k)%has_children = .true.
      return
    end if
  end do
end subroutine

```

```

        end if
    end if
end do
call phs_tree_write (tree)
call msg_bug &
    & (" Missing daughter branch(es) in phase space tree")
end subroutine branch_set_relatives

```

The inverse: this is trivial, fortunately.

### Flip $t$ -channel into $s$ -channel

Flipping the tree is done upwards, beginning from the decay products. First we select a  $t$ -channel branch  $k$ : one which is set, which does have an IN-bit, and which is not an external particle.

Next, we determine the adjacent in-particle (called the 'friend'  $f$  here, since it will provide the reference axis for the angular integration). In addition, we look for the 'mother' and 'sibling' of this particle. If the latter field is empty, we select the (unique) other out-particle which has no mother, calling the internal subroutine `find_orphan`.

The flip is done as follows: We assume that the first daughter  $d$  is an  $s$ -channel line, which is true if the daughters are sorted. This will stay the first daughter. The second one is a  $t$ -channel line; it is exchanged with the 'sibling'  $s$ . The new line which replaces the branch  $k$  is just the sum of  $s$  and  $d$ . In addition, we have to rearrange the relatives of  $s$  and  $d$ , as well of  $f$ .

Finally, we flip 'sibling' and 'friend' and set the new  $s$ -channel branch  $n$  which replaces the  $t$ -channel branch  $k$ . After this is complete, we are ready to execute another flip.

[Although the friend is not needed for the final flip, since it would be an initial particle anyway, we need to know whether we have  $t$ - or  $u$ -channel.]

*(PHS trees: public)*+≡

```
public :: phs_tree_flip_t_to_s_channel
```

*(PHS trees: procedures)*+≡

```

subroutine phs_tree_flip_t_to_s_channel (tree)
    type(phs_tree_t), intent(inout) :: tree
    integer(TC) :: k, f, m, n, d, s
    if (tree%n_in == 2) then
        FLIP: do k=3, tree%mask-1
            if (.not. tree%branch(k)%set) cycle FLIP
            f = iand(k,tree%mask_in)
            if (f==0 .or. f==k) cycle FLIP
            m = tree%branch(k)%mother
            s = tree%branch(k)%sibling
            if (s==0) call find_orphan(s)
            d = tree%branch(k)%daughter(1)
            n = ior(d,s)
            tree%branch(k)%set = .false.
            tree%branch(n)%set = .true.
            tree%branch(n)%origin = k
            tree%branch(n)%daughter(1) = d; tree%branch(d)%mother = n
            tree%branch(n)%daughter(2) = s; tree%branch(s)%mother = n
        end do
    end if
end subroutine

```

```

        tree%branch(n)%has_children = .true.
        tree%branch(d)%sibling = s; tree%branch(s)%sibling = d
        tree%branch(n)%sibling = f; tree%branch(f)%sibling = n
        tree%branch(n)%mother      = m
        tree%branch(f)%mother      = m
        if (m/=0) then
            tree%branch(m)%daughter(1) = n
            tree%branch(m)%daughter(2) = f
        end if
        tree%branch(n)%friend = f
        tree%branch(n)%has_friend = .true.
        tree%branch(n)%firstborn = 2
    end do FLIP
end if
contains
    subroutine find_orphan(s)
        integer(TC) :: s
        do s=1, tree%mask_out
            if (tree%branch(s)%set .and. tree%branch(s)%mother==0) return
        end do
        call phs_tree_write (tree)
        call msg_bug (" Can't flip phase space tree to channel")
    end subroutine find_orphan
end subroutine phs_tree_flip_t_to_s_channel

```

After the tree has been flipped, one may need to determine what has become of a particular *t*-channel branch. This function gives the bincode of the flipped tree. If the original bincode does not contain IN-bits, we leave it as it is.

*(PHS trees: procedures)*+≡

```

function tc_flipped (tree, kt) result (ks)
    type(phs_tree_t), intent(in) :: tree
    integer(TC), intent(in) :: kt
    integer(TC) :: ks
    if (iand (kt, tree%mask_in) == 0) then
        ks = kt
    else
        ks = tree%branch(iand (kt, tree%mask_out))%mother
    end if
end function tc_flipped

```

Scan a tree and make sure that the first daughter has always a smaller code than the second one. Furthermore, delete any `friend` entry in the root branch – this branching has the incoming particle direction as axis anyway. Keep track of reordering by updating `inverted_axis`, `inverted_decay` and `firstborn`.

*(PHS trees: public)*+≡

```

public :: phs_tree_canonicalize

```

*(PHS trees: procedures)*+≡

```

subroutine phs_tree_canonicalize (tree)
    type(phs_tree_t), intent(inout) :: tree
    integer :: n_out
    integer(TC) :: k_out
    call branch_canonicalize (tree%branch(tree%mask_out))

```



```

n_out = tree%n_externals - tree%n_in
k_out = tree%mask_out
if (tree%branch(k_out)%has_friend &
    & .and. tree%branch(k_out)%friend == ibset (0, n_out)) then
    tree%branch(k_out)%inverted_axis = .not.tree%branch(k_out)%inverted_axis
end if
tree%branch(k_out)%has_friend = .false.
tree%branch(k_out)%friend = 0
contains
recursive subroutine branch_canonicalize (b)
    type(phs_branch_t), intent(inout) :: b
    integer(TC) :: d1, d2
    if (b%has_children) then
        d1 = b%daughter(1)
        d2 = b%daughter(2)
        if (d1 > d2) then
            b%daughter(1) = d2
            b%daughter(2) = d1
            b%inverted_decay = .not.b%inverted_decay
            if (b%firstborn /= 0) b%firstborn = 3 - b%firstborn
        end if
        call branch_canonicalize (tree%branch(b%daughter(1)))
        call branch_canonicalize (tree%branch(b%daughter(2)))
    end if
end subroutine branch_canonicalize
end subroutine phs_tree_canonicalize

```

## Mappings

Initialize a mapping for the current tree. This is done while reading from file, so the mapping parameters are read, but applied to the flipped tree. Thus, the size of the array of mappings is given by the number of outgoing particles only.

```

<PHS trees: public>+≡
    public :: phs_tree_init_mapping

<PHS trees: procedures>+≡
    subroutine phs_tree_init_mapping (tree, k, type, pdg, model)
        type(phs_tree_t), intent(inout) :: tree
        integer(TC), intent(in) :: k
        type(string_t), intent(in) :: type
        integer, intent(in) :: pdg
        type(model_t), intent(in), target :: model
        integer(TC) :: kk
        kk = tc_flipped (tree, k)
        call mapping_init (tree%mapping(kk), kk, type, pdg, model)
    end subroutine phs_tree_init_mapping

```

Set the physical parameters for the mapping, using a specific parameter set. Also set the mass sum array.

```

<PHS trees: public>+≡
    public :: phs_tree_set_mapping_parameters

```

```

<PHS trees: procedures>+≡
  subroutine phs_tree_set_mapping_parameters &
    (tree, mapping_defaults, variable_limits)
    type(phs_tree_t), intent(inout) :: tree
    type(mapping_defaults_t), intent(in) :: mapping_defaults
    logical, intent(in) :: variable_limits
    integer(TC) :: k
    do k = 1, tree%n_branches_out
      call mapping_set_parameters &
        (tree%mapping(k), mapping_defaults, variable_limits)
    end do
  end subroutine phs_tree_set_mapping_parameters

```

Return the mapping for the sum of all outgoing particles. This should either be no mapping or a global s-channel mapping.

```

<PHS trees: public>+≡
  public :: phs_tree_assign_s_mapping

<PHS trees: procedures>+≡
  subroutine phs_tree_assign_s_mapping (tree, mapping)
    type(phs_tree_t), intent(in) :: tree
    type(mapping_t), intent(out) :: mapping
    mapping = tree%mapping(tree%mask_out)
  end subroutine phs_tree_assign_s_mapping

```

## Kinematics

Fill the mass sum array, starting from the external particles and working down to the tree root. For each bincode  $k$  we scan the bits in  $k$ ; if only one is set, we take the physical mass of the corresponding external particle; if more than one is set, we sum up the two masses (which we know have already been set).

```

<PHS trees: public>+≡
  public :: phs_tree_set_mass_sum

<PHS trees: procedures>+≡
  subroutine phs_tree_set_mass_sum (tree, flv)
    type(phs_tree_t), intent(inout) :: tree
    type(flavor_t), dimension(:), intent(in) :: flv
    integer(TC) :: k
    integer :: i
    tree%mass_sum = 0
    do k = 1, tree%n_branches_out
      do i = 0, size(flv) - 1
        if (btest(k,i)) then
          if (ibclr(k,i) == 0) then
            tree%mass_sum(k) = flavor_get_mass (flv(i+1))
          else
            tree%mass_sum(k) = &
              tree%mass_sum(ibclr(k,i)) + tree%mass_sum(ibset(0,i))
          end if
        end if
      end do
    end do
  end subroutine phs_tree_set_mass_sum

```

```

        end do
    end do
end subroutine phs_tree_set_mass_sum

```

Set the effective masses and widths. For each non-resonant branch in a tree, the effective mass is equal to the sum of the effective masses of the children (and analogous for the width). External particles have their real mass and width zero. For resonant branches, we insert mass and width from the corresponding mapping.

This routine has `phs_tree_set_mass_sum` and `phs_tree_set_mapping_parameters` as prerequisites.

```

<PHS trees: public>+≡
    public :: phs_tree_set_effective_masses

<PHS trees: procedures>+≡
    subroutine phs_tree_set_effective_masses (tree)
        type(phs_tree_t), intent(inout) :: tree
        integer(TC) :: k
        tree%effective_mass = 0
        tree%effective_width = 0
        call set_masses_x (tree%mask_out)
contains
    recursive subroutine set_masses_x (k)
        integer(TC), intent(in) :: k
        integer(TC) :: k1, k2
        if (tree%branch(k)%has_children) then
            k1 = tree%branch(k)%daughter(1)
            k2 = tree%branch(k)%daughter(2)
            call set_masses_x (k1)
            call set_masses_x (k2)
            if (mapping_is_s_channel (tree%mapping(k))) then
                tree%effective_mass(k) = mapping_get_mass (tree%mapping(k))
                tree%effective_width(k) = mapping_get_width (tree%mapping(k))
            else
                tree%effective_mass(k) = &
                    tree%effective_mass(k1) + tree%effective_mass(k2)
                tree%effective_width(k) = &
                    tree%effective_width(k1) + tree%effective_width(k2)
            end if
        else
            tree%effective_mass(k) = tree%mass_sum(k)
        end if
    end subroutine set_masses_x
end subroutine phs_tree_set_effective_masses

```

Define step mappings, recursively, for the decay products of all intermediate resonances. Step mappings account for the fact that a branch may originate from a resonance, which almost replaces the upper limit on the possible invariant mass. The step mapping implements a smooth cutoff that interpolates between the resonance and the real kinematic limit. The mapping width determines the sharpness of the cutoff.

Step mappings are inserted only for branches that are not mapped otherwise.

At each branch, we record the mass that is effectively available for phase space, by taking the previous limit and subtracting the effective mass of the sibling branch. Widths are added, not subtracted.

If we encounter a resonance decay, we discard the previous limit and replace it by the mass and width of the resonance, also subtracting the sibling branch.

Initially, the limit is zero, so it becomes negative at any branch. Only if there is a resonance, the limit becomes positive. Whenever the limit is positive, and the current branch decays, we activate a step mapping for the current branch.

As a result, step mappings are implemented for all internal lines that originate from an intermediate resonance decay.

The flag `variable_limits` applies to the ultimate limit from the available energy, not to the intermediate resonances whose masses are always fixed.

This routine requires `phs_tree_set_effective_masses`

```

<PHS trees: public>+≡
  public :: phs_tree_set_step_mappings

<PHS trees: procedures>+≡
  subroutine phs_tree_set_step_mappings (tree, exp_type, variable_limits)
    type(phs_tree_t), intent(inout) :: tree
    logical, intent(in) :: exp_type
    logical, intent(in) :: variable_limits
    type(string_t) :: map_str
    integer(TC) :: k
    if (exp_type) then
      map_str = "step_exp"
    else
      map_str = "step_hyp"
    end if
    k = tree%mask_out
    call set_step_mappings_x (k, 0._default, 0._default)
contains
    recursive subroutine set_step_mappings_x (k, m_limit, w_limit)
      integer(TC), intent(in) :: k
      real(default), intent(in) :: m_limit, w_limit
      integer(TC), dimension(2) :: kk
      real(default), dimension(2) :: m, w
      if (tree%branch(k)%has_children) then
        if (m_limit > 0) then
          if (.not. mapping_is_set (tree%mapping(k))) then
            call mapping_init (tree%mapping(k), k, map_str)
            call mapping_set_step_mapping_parameters (tree%mapping(k), &
              m_limit, w_limit, &
              variable_limits)
          end if
        end if
        kk = tree%branch(k)%daughter
        m = tree%effective_mass(kk)
        w = tree%effective_width(kk)
        if (mapping_is_s_channel (tree%mapping(k))) then
          call set_step_mappings_x (kk(1), &
            mapping_get_mass (tree%mapping(k)) - m(2), &
            mapping_get_width (tree%mapping(k)) + w(2))
          call set_step_mappings_x (kk(2), &

```

```

        mapping_get_mass (tree%mapping(k)) - m(1), &
        mapping_get_width (tree%mapping(k)) + w(1))
    else if (m_limit > 0) then
        call set_step_mappings_x (kk(1), &
            m_limit - m(2), &
            w_limit + w(2))
        call set_step_mappings_x (kk(2), &
            m_limit - m(1), &
            w_limit + w(1))
    else
        call set_step_mappings_x (kk(1), &
            - m(2), &
            + w(2))
        call set_step_mappings_x (kk(2), &
            - m(1), &
            + w(1))
    end if
end if
end subroutine set_step_mappings_x
end subroutine phs_tree_set_step_mappings

```

### Structural comparison

This function allows to check whether one tree is the permutation of another one. The permutation is applied to the second tree in the argument list. We do not make up a temporary permuted tree, but compare the two trees directly. The branches are scanned recursively, where for each daughter we check the friend and the mapping as well. Once a discrepancy is found, the recursion is exited immediately.

```

<PHS trees: public>+≡
    public :: phs_tree_equivalent

<PHS trees: procedures>+≡
    function phs_tree_equivalent (t1, t2, perm) result (is_equal)
        type(phs_tree_t), intent(in) :: t1, t2
        type(permutation_t), intent(in) :: perm
        logical :: equal, is_equal
        integer(TC) :: k1, k2, mask_in
        k1 = t1%mask_out
        k2 = t2%mask_out
        mask_in = t1%mask_in
        equal = .true.
        call check (t1%branch(k1), t2%branch(k2), k1, k2)
        is_equal = equal
    contains
        recursive subroutine check (b1, b2, k1, k2)
            type(phs_branch_t), intent(in) :: b1, b2
            integer(TC), intent(in) :: k1, k2
            integer(TC), dimension(2) :: d1, d2, pd2
            integer :: i
            if (.not.b1%has_friend .and. .not.b2%has_friend) then
                equal = .true.
            else if (b1%has_friend .and. b2%has_friend) then

```

```

        equal = (b1%friend == tc_permute (b2%friend, perm, mask_in))
    end if
    if (equal) then
        if (b1%has_children .and. b2%has_children) then
            d1 = b1%daughter
            d2 = b2%daughter
            do i=1, 2
                pd2(i) = tc_permute (d2(i), perm, mask_in)
            end do
            if (d1(1)==pd2(1) .and. d1(2)==pd2(2)) then
                equal = (b1%firstborn == b2%firstborn)
                if (equal) call check &
                    & (t1%branch(d1(1)), t2%branch(d2(1)), d1(1), d2(1))
                if (equal) call check &
                    & (t1%branch(d1(2)), t2%branch(d2(2)), d1(2), d2(2))
            else if (d1(1)==pd2(2) .and. d1(2)==pd2(1)) then
                equal = ( (b1%firstborn == 0 .and. b2%firstborn == 0) &
                    & .or. (b1%firstborn == 3 - b2%firstborn) )
                if (equal) call check &
                    & (t1%branch(d1(1)), t2%branch(d2(2)), d1(1), d2(2))
                if (equal) call check &
                    & (t1%branch(d1(2)), t2%branch(d2(1)), d1(2), d2(1))
            else
                equal = .false.
            end if
        end if
    end if
    if (equal) then
        equal = (t1%mapping(k1) == t2%mapping(k2))
    end if
end subroutine check
end function phs_tree_equivalent

```

Scan two decay trees and determine the correspondence of mass variables, i.e., the permutation that transfers the ordered list of mass variables belonging to the second tree into the first one. Mass variables are assigned beginning from branches and ending at the root.

*(PHS trees: public)*+≡

```
public :: phs_tree_find_msq_permutation
```

*(PHS trees: procedures)*+≡

```

subroutine phs_tree_find_msq_permutation (tree1, tree2, perm2, msq_perm)
    type(phs_tree_t), intent(in) :: tree1, tree2
    type(permutation_t), intent(in) :: perm2
    type(permutation_t), intent(out) :: msq_perm
    type(permutation_t) :: perm1
    integer(TC) :: mask_in, root
    integer(TC), dimension(:), allocatable :: index1, index2
    integer :: i
    allocate (index1 (tree1%n_msq), index2 (tree2%n_msq))
    call permutation_init (perm1, permutation_size (perm2))
    mask_in = tree1%mask_in
    root = tree1%mask_out
    i = 0

```

```

call tree_scan (tree1, root, perm1, index1)
i = 0
call tree_scan (tree2, root, perm2, index2)
call permutation_find (msq_perm, index1, index2)
contains
recursive subroutine tree_scan (tree, k, perm, index)
  type(phas_tree_t), intent(in) :: tree
  integer(TC), intent(in) :: k
  type(permutation_t), intent(in) :: perm
  integer, dimension(:), intent(inout) :: index
  if (tree%branch(k)%has_children) then
    call tree_scan (tree, tree%branch(k)%daughter(1), perm, index)
    call tree_scan (tree, tree%branch(k)%daughter(2), perm, index)
    i = i + 1
    if (i <= size (index)) index(i) = tc_permute (k, perm, mask_in)
  end if
end subroutine tree_scan
end subroutine phs_tree_find_msq_permutation

<PHS trees: public>+≡
public :: phs_tree_find_angle_permutation

<PHS trees: procedures>+≡
subroutine phs_tree_find_angle_permutation &
  (tree1, tree2, perm2, angle_perm, sig2)
  type(phas_tree_t), intent(in) :: tree1, tree2
  type(permutation_t), intent(in) :: perm2
  type(permutation_t), intent(out) :: angle_perm
  logical, dimension(:), allocatable, intent(out) :: sig2
  type(permutation_t) :: perm1
  integer(TC) :: mask_in, root
  integer(TC), dimension(:), allocatable :: index1, index2
  logical, dimension(:), allocatable :: sig1
  integer :: i
  allocate (index1 (tree1%n_angles), index2 (tree2%n_angles))
  allocate (sig1 (tree1%n_angles), sig2 (tree2%n_angles))
  call permutation_init (perm1, permutation_size (perm2))
  mask_in = tree1%mask_in
  root = tree1%mask_out
  i = 0
  call tree_scan (tree1, root, perm1, index1, sig1)
  i = 0
  call tree_scan (tree2, root, perm2, index2, sig2)
  call permutation_find (angle_perm, index1, index2)
contains
recursive subroutine tree_scan (tree, k, perm, index, sig)
  type(phas_tree_t), intent(in) :: tree
  integer(TC), intent(in) :: k
  type(permutation_t), intent(in) :: perm
  integer, dimension(:), intent(inout) :: index
  logical, dimension(:), intent(inout) :: sig
  integer(TC) :: k1, k2, kp
  logical :: s
  if (tree%branch(k)%has_children) then

```

```

        k1 = tree%branch(k)%daughter(1)
        k2 = tree%branch(k)%daughter(2)
        s = (tc_permute(k1, perm, mask_in) < tc_permute(k2, perm, mask_in))
        kp = tc_permute (k, perm, mask_in)
        i = i + 1
        index(i) = kp
        sig(i) = s
        i = i + 1
        index(i) = - kp
        sig(i) = s
        call tree_scan (tree, k1, perm, index, sig)
        call tree_scan (tree, k2, perm, index, sig)
    end if
end subroutine tree_scan
end subroutine phs_tree_find_angle_permutation

```

## 11.5.4 Phase-space evaluation

### Phase-space volume

We compute the phase-space volume recursively, following the same path as for computing other kinematical variables. However, the volume depends just on  $\sqrt{\hat{s}}$ , not on the momentum configuration.

Note: counting branches, we may replace this by a simple formula.

```

<PHS trees: public>+≡
    public :: phs_tree_compute_volume

<PHS trees: procedures>+≡
    subroutine phs_tree_compute_volume (tree, sqrts, volume)
        type(phs_tree_t), intent(in) :: tree
        real(default), intent(in) :: sqrts
        real(default), intent(out) :: volume
        integer(TC) :: k
        k = tree%mask_out
        if (tree%branch(k)%has_children) then
            call compute_volume_x (tree%branch(k), k, volume, .true.)
        else
            volume = 1
        end if
    contains
        recursive subroutine compute_volume_x (b, k, volume, initial)
            type(phs_branch_t), intent(in) :: b
            integer(TC), intent(in) :: k
            real(default), intent(out) :: volume
            logical, intent(in) :: initial
            integer(TC) :: k1, k2
            real(default) :: v1, v2
            k1 = b%daughter(1); k2 = b%daughter(2)
            if (tree%branch(k1)%has_children) then
                call compute_volume_x (tree%branch(k1), k1, v1, .false.)
            else
                v1 = 1
            end if

```



```

        if (tree%branch(k2)%has_children) then
            call compute_volume_x (tree%branch(k2), k2, v2, .false.)
        else
            v2 = 1
        end if
        if (initial) then
            volume = v1 * v2 / (4 * twopi5)
        else
            volume = v1 * v2 * sqrts**2 / (4 * twopi2)
        end if
    end subroutine compute_volume_x
end subroutine phs_tree_compute_volume

```

### Determine momenta

This is done in two steps: First the masses are determined. This step may fail, in which case `ok` is set to `false`. If successful, we generate angles and the actual momenta. The array `decay_p` serves for transferring the individual three-momenta of the daughter particles in their mother rest frame from the mass generation to the momentum generation step.

```

<PHS trees: public>+≡
    public :: phs_tree_compute_momenta_from_x

<PHS trees: procedures>+≡
    subroutine phs_tree_compute_momenta_from_x &
        (tree, prt, factor, volume, sqrts, x, ok)
        type(phs_tree_t), intent(inout) :: tree
        type(phs_prt_t), dimension(:), intent(inout) :: prt
        real(default), intent(out) :: factor, volume
        real(default), intent(in) :: sqrts
        real(default), dimension(:), intent(in) :: x
        logical, intent(out) :: ok
        real(default), dimension(tree%mask_out) :: decay_p
        integer :: n1, n2
        n1 = tree%n_msq
        n2 = n1 + tree%n_angles
        call phs_tree_set_msq &
            (tree, prt, factor, volume, decay_p, sqrts, x(1:n1), ok)
        if (ok) call phs_tree_set_angles &
            (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
    end subroutine phs_tree_compute_momenta_from_x

```

Mass generation is done recursively. The `ok` flag causes the filled tree to be discarded if set to `false`. This happens if a three-momentum turns out to be imaginary, indicating impossible kinematics. The index `ix` tells us how far we have used up the input array `x`.

```

<PHS trees: procedures>+≡
    subroutine phs_tree_set_msq &
        (tree, prt, factor, volume, decay_p, sqrts, x, ok)
        type(phs_tree_t), intent(inout) :: tree
        type(phs_prt_t), dimension(:), intent(inout) :: prt
        real(default), intent(out) :: factor, volume

```

```

real(default), dimension(:), intent(out) :: decay_p
real(default), intent(in) :: sqrts
real(default), dimension(:), intent(in) :: x
logical, intent(out) :: ok
integer :: ix
integer(TC) :: k
real(default) :: m_tot
ok = .true.
ix = 1
k = tree%mask_out
m_tot = tree%mass_sum(k)
decay_p(k) = 0.
if (m_tot < sqrts .or. k == 1) then
  if (tree%branch(k)%has_children) then
    call set_msq_x (tree%branch(k), k, factor, volume, .true.)
  else
    factor = 1
    volume = 1
  end if
else
  ok = .false.
end if
contains
recursive subroutine set_msq_x (b, k, factor, volume, initial)
  type(phs_branch_t), intent(in) :: b
  integer(TC), intent(in) :: k
  real(default), intent(out) :: factor, volume
  logical, intent(in) :: initial
  real(default) :: msq, m, m_min, m_max, m1, m2, msq1, msq2, lda, rlda
  integer(TC) :: k1, k2
  real(default) :: f1, f2, v1, v2
  k1 = b%daughter(1); k2 = b%daughter(2)
  if (tree%branch(k1)%has_children) then
    call set_msq_x (tree%branch(k1), k1, f1, v1, .false.)
    if (.not.ok) return
  else
    f1 = 1; v1 = 1
  end if
  if (tree%branch(k2)%has_children) then
    call set_msq_x (tree%branch(k2), k2, f2, v2, .false.)
    if (.not.ok) return
  else
    f2 = 1; v2 = 1
  end if
  m_min = tree%mass_sum(k)
  if (initial) then
    msq = sqrts**2
    m = sqrts
    m_max = sqrts
    factor = f1 * f2
    volume = v1 * v2 / (4 * twopi5)
  else
    m_max = sqrts - m_tot + m_min
    call mapping_compute_msq_from_x &

```

```

        (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
        x(ix)); ix = ix + 1
    if (msq >= 0) then
        m = sqrt (msq)
        factor = f1 * f2 * factor
        volume = v1 * v2 * sqrts**2 / (4 * twopi2)
        call phs_prt_set_msq (prt(k), msq)
        call phs_prt_set_defined (prt(k))
    else
        ok = .false.
    end if
end if
if (ok) then
    msq1 = phs_prt_get_msq (prt(k1)); m1 = sqrt (msq1)
    msq2 = phs_prt_get_msq (prt(k2)); m2 = sqrt (msq2)
    lda = lambda (msq, msq1, msq2)
    if (lda > 0 .and. m > m1 + m2 .and. m <= m_max) then
        rlda = sqrt (lda)
        decay_p(k1) = rlda / (2*m)
        decay_p(k2) = - decay_p(k1)
        factor = rlda / msq * factor
    else
        ok = .false.
    end if
end if
end subroutine set_msq_x
end subroutine phs_tree_set_msq

```

The heart of phase space generation: Now we have the invariant masses, let us generate angles. At each branch, we take a Lorentz transformation and augment it by a boost to the current particle rest frame, and by rotations  $\phi$  and  $\theta$  around the  $z$  and  $y$  axis, respectively. This transformation is passed down to the daughter particles, if present.

*(PHS trees: procedures)+≡*

```

subroutine phs_tree_set_angles (tree, prt, factor, decay_p, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(inout) :: prt
    real(default), intent(inout) :: factor
    real(default), dimension(:), intent(in) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(in) :: x
    integer :: ix
    integer(TC) :: k
    ix = 1
    k = tree%mask_out
    call set_angles_x (tree%branch(k), k)
contains
    recursive subroutine set_angles_x (b, k, L0)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        type(lorentz_transformation_t), intent(in), optional :: L0
        real(default) :: m, msq, ct, st, phi, f, E, p, bg
        type(lorentz_transformation_t) :: L, LL
    end subroutine set_angles_x
end subroutine phs_tree_set_angles

```

```

integer(TC) :: k1, k2
type(vector3_t) :: axis
p = decay_p(k)
msq = phs_prt_get_msq (prt(k)); m = sqrt (msq)
E = sqrt (msq + p**2)
if (present (L0)) then
    call phs_prt_set_momentum (prt(k), L0 * vector4_moving (E,p,3))
else
    call phs_prt_set_momentum (prt(k), vector4_moving (E,p,3))
end if
call phs_prt_set_defined (prt(k))
if (b%has_children) then
    k1 = b%daughter(1)
    k2 = b%daughter(2)
    if (m > 0) then
        bg = p / m
    else
        bg = 0
    end if
    phi = x(ix) * twopi; ix = ix + 1
    call mapping_compute_ct_from_x &
        (tree%mapping(k), sqrts**2, ct, st, f, x(ix)); ix = ix + 1
    factor = factor * f
    if (.not. b%has_friend) then
!       L = boost (bg,3) * rotation (phi,3) * rotation (ct,st,2)
       L = LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), bg)
    else
        LL = boost (-bg,3); if (present (L0)) LL = LL * inverse(L0)
        axis = space_part ( &
            LL * phs_prt_get_momentum (prt(tree%branch(k)%friend)) )
        L = boost(bg,3) * rotation_to_2nd (vector3_canonical(3), axis) &
!         & * rotation(phi,3) * rotation(ct,st,2)
        * LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), 0._default)
    end if
    if (present (L0)) L = L0 * L
    call set_angles_x (tree%branch(k1), k1, L)
    call set_angles_x (tree%branch(k2), k2, L)
end if
end subroutine set_angles_x
end subroutine phs_tree_set_angles

```

## Recover random numbers

For the other channels we want to compute the random numbers that would have generated the momenta that we already know.

*<PHS trees: public>+≡*

```
public :: phs_tree_compute_x_from_momenta
```

*<PHS trees: procedures>+≡*

```

subroutine phs_tree_compute_x_from_momenta (tree, prt, factor, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(in) :: prt
    real(default), intent(out) :: factor

```

```

real(default), intent(in) :: sqrts
real(default), dimension(:), intent(inout) :: x
real(default), dimension(tree%mask_out) :: decay_p
integer :: n1, n2
n1 = tree%n_msq
n2 = n1 + tree%n_angles
call phs_tree_get_msq &
    (tree, prt, factor, decay_p, sqrts, x(1:n1))
call phs_tree_get_angles &
    (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
end subroutine phs_tree_compute_x_from_momenta

```

The inverse operation follows exactly the same steps. The tree is `inout` because it contains mappings whose parameters can be reset when the mapping is applied.

*(PHS trees: procedures)* +=

```

subroutine phs_tree_get_msq (tree, prt, factor, decay_p, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(in) :: prt
    real(default), intent(out) :: factor
    real(default), dimension(:), intent(out) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(inout) :: x
    integer :: ix
    integer(TC) :: k
    real(default) :: m_tot
    ix = 1
    k = tree%mask_out
    m_tot = tree%mass_sum(k)
    decay_p(k) = 0.
    if (tree%branch(k)%has_children) then
        call get_msq_x (tree%branch(k), k, factor, .true.)
    else
        factor = 1
    end if
contains
    recursive subroutine get_msq_x (b, k, factor, initial)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(out) :: factor
        logical, intent(in) :: initial
        real(default) :: msq, m, m_min, m_max, msq1, msq2, lda, rlda
        integer(TC) :: k1, k2
        real(default) :: f1, f2
        k1 = b%daughter(1); k2 = b%daughter(2)
        if (tree%branch(k1)%has_children) then
            call get_msq_x (tree%branch(k1), k1, f1, .false.)
        else
            f1 = 1
        end if
        if (tree%branch(k2)%has_children) then
            call get_msq_x (tree%branch(k2), k2, f2, .false.)
        else

```

```

        f2 = 1
    end if
    m_min = tree%mass_sum(k)
    m_max = sqrts - m_tot + m_min
    msq = phs_prt_get_msq (prt(k)); m = sqrt (msq)
    if (initial) then
        factor = f1 * f2
    else
        call mapping_compute_x_from_msq &
            (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
                x(ix)); ix = ix + 1
        factor = f1 * f2 * factor
    end if
    msq1 = phs_prt_get_msq (prt(k1))
    msq2 = phs_prt_get_msq (prt(k2))
    lda = lambda (msq, msq1, msq2)
    if (lda > 0) then
        rlda = sqrt (lda)
        decay_p(k1) = rlda / (2 * m)
        decay_p(k2) = - decay_p(k1)
        factor = rlda / msq * factor
    else
        decay_p(k1) = 0
        decay_p(k2) = 0
        factor = 0
    end if
end subroutine get_msq_x
end subroutine phs_tree_get_msq

```

This subroutine is the most time-critical part of the whole program. Therefore, we do not exactly parallel the angle generation routine above but make sure that things get evaluated only if they are really needed, at the expense of readability. Particularly important is to have as few multiplications of Lorentz transformations as possible.

(*PHS trees: procedures*) $\vdash$

```

subroutine phs_tree_get_angles (tree, prt, factor, decay_p, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(in) :: prt
    real(default), intent(inout) :: factor
    real(default), dimension(:), intent(in) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(out) :: x
    integer :: ix
    integer(TC) :: k
    ix = 1
    k = tree%mask_out
    if (tree%branch(k)%has_children) call get_angles_x (tree%branch(k), k)
contains
    recursive subroutine get_angles_x (b, k, ct0, st0, phi0, L0)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(in), optional :: ct0, st0, phi0
        type(lorentz_transformation_t), intent(in), optional :: L0
    end subroutine get_angles_x
end subroutine phs_tree_get_angles

```

```

real(default) :: cp0, sp0, m, msq, ct, st, phi, bg, f
type(lorentz_transformation_t) :: L, LL
type(vector4_t) :: p1, pf
type(vector3_t) :: n, axis
integer(TC) :: k1, k2, kf
logical :: has_friend, need_L
k1 = b%daughter(1)
k2 = b%daughter(2)
kf = b%friend
has_friend = b%has_friend
if (present(L0)) then
    p1 = L0 * phs_prt_get_momentum (prt(k1))
    if (has_friend) pf = L0 * phs_prt_get_momentum (prt(kf))
else
    p1 = phs_prt_get_momentum (prt(k1))
    if (has_friend) pf = phs_prt_get_momentum (prt(kf))
end if
if (present(phi0)) then
    cp0 = cos (phi0)
    sp0 = sin (phi0)
end if
msq = phs_prt_get_msq (prt(k)); m = sqrt (msq)
if (m > 0) then
    bg = decay_p(k) / m
else
    bg = 0
end if
if (has_friend) then
    if (present (phi0)) then
        axis = axis_from_p_r3_r2_b3 (pf, cp0, -sp0, ct0, -st0, -bg)
        LL = rotation_to_2nd (axis, vector3_canonical (3)) &
            * LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
    else
        axis = axis_from_p_b3 (pf, -bg)
        LL = rotation_to_2nd (axis, vector3_canonical(3))
        if (bg /= 0) LL = LL * boost(-bg, 3)
    end if
    n = space_part (LL * p1)
else if (present (phi0)) then
    n = axis_from_p_r3_r2_b3 (p1, cp0, -sp0, ct0, -st0, -bg)
else
    n = axis_from_p_b3 (p1, -bg)
end if
phi = azimuthal_angle (n)
x(ix) = phi / twopi; ix = ix + 1
ct = polar_angle_ct (n)
st = sqrt (1 - ct**2)
call mapping_compute_x_from_ct (tree%mapping(k), sqrts**2, ct, f, &
    x(ix)); ix = ix + 1
factor = factor * f
if (tree%branch(k1)%has_children .or. tree%branch(k2)%has_children) then
    need_L = .true.
    if (has_friend) then
        if (present (L0)) then

```

```

        L = LL * L0
    else
        L = LL
    end if
else if (present (L0)) then
    L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg) * L0
else if (present (phi0)) then
    L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
else if (bg /= 0) then
    L = boost(-bg, 3)
else
    need_L = .false.
end if
if (need_L) then
    if (tree%branch(k1)%has_children) &
        call get_angles_x (tree%branch(k1), k1, ct, st, phi, L)
    if (tree%branch(k2)%has_children) &
        call get_angles_x (tree%branch(k2), k2, ct, st, phi, L)
else
    if (tree%branch(k1)%has_children) &
        call get_angles_x (tree%branch(k1), k1, ct, st, phi)
    if (tree%branch(k2)%has_children) &
        call get_angles_x (tree%branch(k2), k2, ct, st, phi)
end if
end if
end subroutine get_angles_x
end subroutine phs_tree_get_angles

```

### Auxiliary stuff

This calculates all momenta that are not yet known by summing up daughter particle momenta. The external particles must be known. Only composite particles not yet known are calculated.

```

<PHS trees: public>+≡
    public :: phs_tree_combine_particles

<PHS trees: procedures>+≡
    subroutine phs_tree_combine_particles (tree, prt)
        type(phs_tree_t), intent(in) :: tree
        type(phs_prt_t), dimension(:), intent(inout) :: prt
        call combine_particles_x (tree%mask_out)
contains
    recursive subroutine combine_particles_x (k)
        integer(TC), intent(in) :: k
        integer :: k1, k2
        if (tree%branch(k)%has_children) then
            k1 = tree%branch(k)%daughter(1); k2 = tree%branch(k)%daughter(2)
            call combine_particles_x (k1)
            call combine_particles_x (k2)
            if (.not. prt(k)%defined) then
                call phs_prt_combine (prt(k), prt(k1), prt(k2))
            end if
        end if
    end if
end if

```



```

        end subroutine combine_particles_x
    end subroutine phs_tree_combine_particles

```

The previous routine is to be evaluated at runtime. Instead of scanning trees, we can as well set up a multiplication table. This is generated here. Note that the table is `intent(out)`.

```

<PHS trees: public>+≡
    public :: phs_tree_setup_prt_combinations

<PHS trees: procedures>+≡
    subroutine phs_tree_setup_prt_combinations (tree, comb)
        type(phs_tree_t), intent(in) :: tree
        integer, dimension(:,,:), intent(out) :: comb
        comb = 0
        call setup_prt_combinations_x (tree%mask_out)
    contains
        recursive subroutine setup_prt_combinations_x (k)
            integer(TC), intent(in) :: k
            integer, dimension(2) :: kk
            if (tree%branch(k)%has_children) then
                kk = tree%branch(k)%daughter
                call setup_prt_combinations_x (kk(1))
                call setup_prt_combinations_x (kk(2))
                comb(:,k) = kk
            end if
        end subroutine setup_prt_combinations_x
    end subroutine phs_tree_setup_prt_combinations

```

## 11.6 The phase-space forest

Simply stated, a phase-space forest is a collection of phase-space trees. More precisely, a `phs_forest` object contains all parameterizations of phase space that `WHIZARD` will use for a single hard process, prepared in the form of `phs_tree` objects. This is suitable for evaluation by the `VAMP` integration package: each parameterization (tree) is a valid channel in the multi-channel adaptive integration, and each variable in a tree corresponds to an integration dimension, defined by an appropriate mapping of the  $(0, 1)$  interval to the allowed range of the integration variable.

The trees are grouped in groves. The trees (integration channels) within a grove share a common weight, assuming that they are related by some approximate symmetry.

Trees/channels that are related by an exact symmetry are connected by an array of equivalences; each equivalence object holds the data that relate one channel to another.

The phase-space setup, i.e., the detailed structure of trees and forest, are read from a file. Therefore, this module also contains the syntax definition and the parser needed for interpreting this file.

```
(phs_forests.f90)≡  
  <File header>  
  
  module phs_forests  
  
    <Use kinds>  
    use kinds, only: TC !NODEP!  
    <Use strings>  
    <Use file utils>  
    use diagnostics !NODEP!  
    use lorentz !NODEP!  
    ! use vamp_equivalences !NODEP!  
    use permutations  
    use ifiles  
    use syntax_rules  
    use lexers  
    use parser  
    use models  
    use flavors  
    use interactions  
  
    use phs_base  
    use mappings  
    use phs_trees  
  
    <Standard module head>  
  
    <PHS forests: public>  
  
    <PHS forests: types>  
  
    <PHS forests: interfaces>
```

*<PHS forests: variables>*

contains

*<PHS forests: procedures>*

end module phs\_forests

### 11.6.1 Phase-space setup parameters

This transparent container holds the parameters that the algorithm needs for phase-space setup, with reasonable defaults.

The threshold mass (for considering a particle as effectively massless) is specified separately for s- and t-channel. The default is to treat  $W$  and  $Z$  bosons as massive in the s-channel, but as massless in the t-channel. The  $b$ -quark is treated always massless, the  $t$ -quark always massive.

*<PHS forests: public>*≡

public :: phs\_parameters\_t

*<PHS forests: types>*≡

```
type :: phs_parameters_t
  real(default) :: sqrts = 0
  real(default) :: m_threshold_s = 50._default
  real(default) :: m_threshold_t = 100._default
  integer :: off_shell = 1
  integer :: t_channel = 2
  logical :: keep_nonresonant = .true.
end type phs_parameters_t
```

Write phase-space parameters to file.

*<PHS forests: public>*+≡

public :: phs\_parameters\_write

*<PHS forests: procedures>*≡

```
subroutine phs_parameters_write (phs_par, unit)
  type(phs_parameters_t), intent(in) :: phs_par
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A,ES19.12)") "sqrts          = ", phs_par%sqrts
  write (u, "(3x,A,ES19.12)") "m_threshold_s = ", phs_par%m_threshold_s
  write (u, "(3x,A,ES19.12)") "m_threshold_t = ", phs_par%m_threshold_t
  write (u, "(3x,A,I0)") "off_shell = ", phs_par%off_shell
  write (u, "(3x,A,I0)") "t_channel = ", phs_par%t_channel
  write (u, "(3x,A,L1)") "keep_nonresonant = ", phs_par%keep_nonresonant
end subroutine phs_parameters_write
```

Read phase-space parameters from file.

*<PHS forests: public>*+≡

public :: phs\_parameters\_read

```

(PHS forests: procedures)+≡
subroutine phs_parameters_read (phs_par, unit)
  type(phs_parameters_t), intent(out) :: phs_par
  integer, intent(in) :: unit
  character(20) :: dummy
  character :: equals
  read (unit, *) dummy, equals, phs_par%sqrts
  read (unit, *) dummy, equals, phs_par%m_threshold_s
  read (unit, *) dummy, equals, phs_par%m_threshold_t
  read (unit, *) dummy, equals, phs_par%off_shell
  read (unit, *) dummy, equals, phs_par%t_channel
  read (unit, *) dummy, equals, phs_par%keep_nonresonant
end subroutine phs_parameters_read

```

Comparison.

```

(PHS forests: interfaces)≡
interface operator(==)
  module procedure phs_parameters_eq
end interface
interface operator(/=)
  module procedure phs_parameters_ne
end interface

(PHS forests: procedures)+≡
function phs_parameters_eq (phs_par1, phs_par2) result (equal)
  logical :: equal
  type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
  equal = phs_par1%sqrts == phs_par2%sqrts &
    .and. phs_par1%m_threshold_s == phs_par2%m_threshold_s &
    .and. phs_par1%m_threshold_t == phs_par2%m_threshold_t &
    .and. phs_par1%off_shell == phs_par2%off_shell &
    .and. phs_par1%t_channel == phs_par2%t_channel &
    .and.(phs_par1%keep_nonresonant .eqv. phs_par2%keep_nonresonant)
end function phs_parameters_eq

function phs_parameters_ne (phs_par1, phs_par2) result (ne)
  logical :: ne
  type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
  ne = phs_par1%sqrts /= phs_par2%sqrts &
    .or. phs_par1%m_threshold_s /= phs_par2%m_threshold_s &
    .or. phs_par1%m_threshold_t /= phs_par2%m_threshold_t &
    .or. phs_par1%off_shell /= phs_par2%off_shell &
    .or. phs_par1%t_channel /= phs_par2%t_channel &
    .or.(phs_par1%keep_nonresonant .neqv. phs_par2%keep_nonresonant)
end function phs_parameters_ne

```

## 11.6.2 Equivalences

This type holds information about equivalences between phase-space trees. We make a linked list, where each node contains the two trees which are equivalent and the corresponding permutation of external particles. Two more arrays are

to be filled: The permutation of mass variables and the permutation of angular variables, where the signature indicates a necessary exchange of daughter branches.

```

(PHS forests: types)+≡
  type :: equivalence_t
  private
    integer :: left, right
    type(permutation_t) :: perm
    type(permutation_t) :: msq_perm, angle_perm
    logical, dimension(:), allocatable :: angle_sig
    type(equivalence_t), pointer :: next => null ()
  end type equivalence_t

```

```

(PHS forests: types)+≡
  type :: equivalence_list_t
  private
    integer :: length = 0
    type(equivalence_t), pointer :: first => null ()
    type(equivalence_t), pointer :: last => null ()
  end type equivalence_list_t

```

Append an equivalence to the list

```

(PHS forests: procedures)+≡
  subroutine equivalence_list_add (eql, left, right, perm)
    type(equivalence_list_t), intent(inout) :: eql
    integer, intent(in) :: left, right
    type(permutation_t), intent(in) :: perm
    type(equivalence_t), pointer :: eq
    allocate (eq)
    eq%left = left
    eq%right = right
    eq%perm = perm
    if (associated (eql%last)) then
      eql%last%next => eq
    else
      eql%first => eq
    end if
    eql%last => eq
    eql%length = eql%length + 1
  end subroutine equivalence_list_add

```

Delete the list contents. Has to be pure because it is called from an elemental subroutine.

```

(PHS forests: procedures)+≡
  pure subroutine equivalence_list_final (eql)
    type(equivalence_list_t), intent(inout) :: eql
    type(equivalence_t), pointer :: eq
    do while (associated (eql%first))
      eq => eql%first
      eql%first => eql%first%next
      deallocate (eq)
    end do
  end subroutine

```

```

    eql%last => null ()
    eql%length = 0
end subroutine equivalence_list_final

```

Make a deep copy of the equivalence list. This allows for deep copies of groves and forests.

```

<PHS forests: interfaces>+≡
  interface assignment(=)
    module procedure equivalence_list_assign
  end interface

<PHS forests: procedures>+≡
  subroutine equivalence_list_assign (eql_out, eql_in)
    type(equivalence_list_t), intent(out) :: eql_out
    type(equivalence_list_t), intent(in) :: eql_in
    type(equivalence_t), pointer :: eq, eq_copy
    eq => eql_in%first
    do while (associated (eq))
      allocate (eq_copy)
      eq_copy = eq
      eq_copy%next => null ()
      if (associated (eql_out%first)) then
        eql_out%last%next => eq_copy
      else
        eql_out%first => eq_copy
      end if
      eql_out%last => eq_copy
      eq => eq%next
    end do
  end subroutine equivalence_list_assign

```

The number of list entries

```

<PHS forests: procedures>+≡
  elemental function equivalence_list_length (eql) result (length)
    integer :: length
    type(equivalence_list_t), intent(in) :: eql
    length = eql%length
  end function equivalence_list_length

```

Recursively write the equivalences list

```

<PHS forests: procedures>+≡
  subroutine equivalence_list_write (eql, unit)
    type(equivalence_list_t), intent(in) :: eql
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    if (associated (eql%first)) then
      call equivalence_write_rec (eql%first, u)
    else
      write (u, *) " [empty]"
    end if
  contains

```

```

recursive subroutine equivalence_write_rec (eq, u)
  type(equivalence_t), intent(in) :: eq
  integer, intent(in) :: u
  integer :: i
  write (u, "(3x,A,1x,I0,1x,I0,2x,A)", advance="no") &
    "Equivalence:", eq%left, eq%right, "Final state permutation:"
  call permutation_write (eq%perm, u)
  write (u, "(1x,12x,1x,A,1x)", advance="no") &
    "      msq permutation:  "
  call permutation_write (eq%msq_perm, u)
  write (u, "(1x,12x,1x,A,1x)", advance="no") &
    "      angle permutation:"
  call permutation_write (eq%angle_perm, u)
  write (u, "(1x,12x,1x,26x)", advance="no")
  do i = 1, size (eq%angle_sig)
    if (eq%angle_sig(i)) then
      write (u, "(1x,A)", advance="no") "+"
    else
      write (u, "(1x,A)", advance="no") "-"
    end if
  end do
  write (u, *)
  if (associated (eq%next)) call equivalence_write_rec (eq%next, u)
end subroutine equivalence_write_rec
end subroutine equivalence_list_write

```

### 11.6.3 Groves

A grove is a group of trees (phase-space channels) that share a common weight in the integration. Within a grove, channels can be declared equivalent, so they also share their integration grids (up to symmetries). The grove contains a list of equivalences. The `tree_count_offset` is the total number of trees of the preceding groves; when the trees are counted per forest (integration channels), the offset has to be added to all tree indices.

```

<PHS forests: types>+≡
  type :: phs_grove_t
  private
  integer :: tree_count_offset
  type(phs_tree_t), dimension(:), allocatable :: tree
  type(equivalence_list_t) :: equivalence_list
end type phs_grove_t

```

Call `phs_tree_init` which is also elemental:

```

<PHS forests: procedures>+≡
  elemental subroutine phs_grove_init &
    (grove, n_trees, n_in, n_out, n_masses, n_angles)
  type(phs_grove_t), intent(inout) :: grove
  integer, intent(in) :: n_trees, n_in, n_out, n_masses, n_angles
  grove%tree_count_offset = 0
  allocate (grove%tree (n_trees))
  call phs_tree_init (grove%tree, n_in, n_out, n_masses, n_angles)

```

```
end subroutine phs_grove_init
```

The trees do not have pointer components, thus no call to `phs_tree_final`:

```
(PHS forests: procedures)+≡
  elemental subroutine phs_grove_final (grove)
    type(phs_grove_t), intent(inout) :: grove
    deallocate (grove%tree)
    call equivalence_list_final (grove%equivalence_list)
  end subroutine phs_grove_final
```

Deep copy.

```
(PHS forests: interfaces)+≡
  interface assignment(=)
    module procedure phs_grove_assign0
    module procedure phs_grove_assign1
  end interface

(PHS forests: procedures)+≡
  subroutine phs_grove_assign0 (grove_out, grove_in)
    type(phs_grove_t), intent(out) :: grove_out
    type(phs_grove_t), intent(in) :: grove_in
    grove_out%tree_count_offset = grove_in%tree_count_offset
    if (allocated (grove_in%tree)) then
      allocate (grove_out%tree (size (grove_in%tree)))
      grove_out%tree = grove_in%tree
    end if
    grove_out%equivalence_list = grove_in%equivalence_list
  end subroutine phs_grove_assign0

  subroutine phs_grove_assign1 (grove_out, grove_in)
    type(phs_grove_t), dimension(:), intent(out) :: grove_out
    type(phs_grove_t), dimension(:), intent(in) :: grove_in
    integer :: i
    do i = 1, size (grove_in)
      call phs_grove_assign0 (grove_out(i), grove_in(i))
    end do
  end subroutine phs_grove_assign1
```

Get the global (s-channel) mappings. Implemented as a subroutine which returns an array (slice).

```
(PHS forests: procedures)+≡
  subroutine phs_grove_assign_s_mappings (grove, mapping)
    type(phs_grove_t), intent(in) :: grove
    type(mapping_t), dimension(:), intent(out) :: mapping
    integer :: i
    if (size (mapping) == size (grove%tree)) then
      do i = 1, size (mapping)
        call phs_tree_assign_s_mapping (grove%tree(i), mapping(i))
      end do
    else
      call msg_bug ("phs_grove_assign_s_mappings: array size mismatch")
    end if
```



```
end subroutine phs_grove_assign_s_mappings
```

#### 11.6.4 The forest type

This is a collection of trees and associated particles. In a given tree, each branch code corresponds to a particle in the `prt` array. Furthermore, we have an array of mass sums which is independent of the decay tree and of the particular event. The mappings directly correspond to the decay trees, and the decay groves collect the trees in classes. The permutation list consists of all permutations of outgoing particles that map the decay forest onto itself.

The particle codes `flv` (one for each external particle) are needed for determining masses and such. The trees and associated information are collected in the `grove` array, together with a lookup table that associates tree indices to groves. Finally, the `prt` array serves as workspace for phase-space evaluation.

The `prt_combination` is a list of index pairs, namely the particle momenta pairs that need to be combined in order to provide all momentum combinations that the phase-space trees need to know.

```
<PHS forests: public>+≡
  public :: phs_forest_t

<PHS forests: types>+≡
  type :: phs_forest_t
    private
    integer :: n_in, n_out, n_tot
    integer :: n_masses, n_angles, n_dimensions
    integer :: n_trees, n_equivalences
    type(flavor_t), dimension(:), allocatable :: flv
    type(phs_grove_t), dimension(:), allocatable :: grove
    integer, dimension(:), allocatable :: grove_lookup
    type(phs_prt_t), dimension(:), allocatable :: prt_in
    type(phs_prt_t), dimension(:), allocatable :: prt_out
    type(phs_prt_t), dimension(:), allocatable :: prt
    integer(TC), dimension(:,:), allocatable :: prt_combination
    type(mapping_t), dimension(:), allocatable :: s_mapping
  contains
    <PHS forests: phs forest: TBP>
  end type phs_forest_t
```

The initialization merely allocates memory. We have to know how many trees there are in each grove, so we can initialize everything. The number of groves is the size of the `n_tree` array.

In the `grove_lookup` table we store the grove index that belongs to each absolute tree index. The difference between the absolute index and the relative (to the grove) index is stored, for each grove, as `tree_count_offset`.

The particle array is allocated according to the total number of branches each tree has, but not filled.

```
<PHS forests: public>+≡
  public :: phs_forest_init
```

*(PHS forests: procedures)*+≡

```

subroutine phs_forest_init (forest, n_tree, n_in, n_out)
  type(phs_forest_t), intent(inout) :: forest
  integer, dimension(:), intent(in) :: n_tree
  integer, intent(in) :: n_in, n_out
  integer :: g, count, k_root
  forest%n_in = n_in
  forest%n_out = n_out
  forest%n_tot = n_in + n_out
  forest%n_masses = max (n_out - 2, 0)
  forest%n_angles = max (2*n_out - 2, 0)
  forest%n_dimensions = forest%n_masses + forest%n_angles
  forest%n_trees = sum (n_tree)
  forest%n_equivalences = 0
  allocate (forest%grove (size (n_tree)))
  call phs_grove_init &
    (forest%grove, n_tree, n_in, n_out, forest%n_masses, forest%n_angles)
  allocate (forest%grove_lookup (forest%n_trees))
  count = 0
  do g = 1, size (forest%grove)
    forest%grove(g)%tree_count_offset = count
    forest%grove_lookup (count+1:count+n_tree(g)) = g
    count = count + n_tree(g)
  end do
  allocate (forest%prt_in (n_in))
  allocate (forest%prt_out (n_out))
  k_root = 2**forest%n_tot - 1
  allocate (forest%prt (k_root))
  allocate (forest%prt_combination (2, k_root))
  allocate (forest%s_mapping (forest%n_trees))
end subroutine phs_forest_init

```

Assign the global (s-channel) mappings.

*(PHS forests: public)*+≡

```

public :: phs_forest_set_s_mappings

```

*(PHS forests: procedures)*+≡

```

subroutine phs_forest_set_s_mappings (forest)
  type(phs_forest_t), intent(inout) :: forest
  integer :: g, i0, i1, n
  do g = 1, size (forest%grove)
    call phs_forest_get_grove_bounds (forest, g, i0, i1, n)
    call phs_grove_assign_s_mappings &
      (forest%grove(g), forest%s_mapping(i0:i1))
  end do
end subroutine phs_forest_set_s_mappings

```

The grove finalizer is called because it contains the equivalence list:

*(PHS forests: public)*+≡

```

public :: phs_forest_final

```

*(PHS forests: procedures)*+≡

```

subroutine phs_forest_final (forest)
  type(phs_forest_t), intent(inout) :: forest

```

```

    if (allocated (forest%grove)) then
        call phs_grove_final (forest%grove)
        deallocate (forest%grove)
    end if
    if (allocated (forest%grove_lookup)) deallocate (forest%grove_lookup)
    if (allocated (forest%prt)) deallocate (forest%prt)
    if (allocated (forest%s_mapping)) deallocate (forest%s_mapping)
end subroutine phs_forest_final

```

## 11.6.5 Screen output

Write the particles that are non-null, then the trees which point to them:

```

<PHS forests: public>+=
    public :: phs_forest_write

<PHS forests: phs forest: TBP>=
    procedure :: write => phs_forest_write

<PHS forests: procedures>+=
    subroutine phs_forest_write (forest, unit)
        class(phs_forest_t), intent(in) :: forest
        integer, intent(in), optional :: unit
        integer :: u
        integer :: i, g, k
        u = output_unit (unit); if (u < 0) return
        write (u, "(1x,A)") "Phase space forest:"
        write (u, "(3x,A,I0)") "n_in  = ", forest%n_in
        write (u, "(3x,A,I0)") "n_out = ", forest%n_out
        write (u, "(3x,A,I0)") "n_tot = ", forest%n_tot
        write (u, "(3x,A,I0)") "n_masses = ", forest%n_masses
        write (u, "(3x,A,I0)") "n_angles = ", forest%n_angles
        write (u, "(3x,A,I0)") "n_dim  = ", forest%n_dimensions
        write (u, "(3x,A,I0)") "n_trees = ", forest%n_trees
        write (u, "(3x,A,I0)") "n_equiv = ", forest%n_equivalences
        write (u, "(3x,A)", advance="no") "flavors ="
        if (allocated (forest%flv)) then
            do i = 1, size (forest%flv)
                write (u, "(1x,I0)", advance="no") flavor_get_pdg (forest%flv(i))
            end do
            write (u, "(A)")
        else
            write (u, "(1x,A)") "[empty]"
        end if
        write (u, "(1x,A)") "Particle combinations:"
        if (allocated (forest%prt_combination)) then
            do k = 1, size (forest%prt_combination, 2)
                if (forest%prt_combination(1, k) /= 0) then
                    write (u, "(3x,I0,1x,'<=',1x,I0,1x,'+',1x,I0)") &
                        k, forest%prt_combination(:,k)
                end if
            end do
        else
            write (u, "(3x,A)") " [empty]"
        end if
    end subroutine

```

```

end if
write (u, "(1x,A)") "Groves and trees:"
if (allocated (forest%grove)) then
    do g = 1, size (forest%grove)
        write (u, "(3x,A,1x,I0)") "Grove    ", g
        call phs_grove_write (forest%grove(g), unit)
    end do
else
    write (u, "(3x,A)") " [empty]"
end if
write (u, "(1x,A,I0)") "Total number of equivalences: ", &
    forest%n_equivalences
write (u, "(A)")
write (u, "(1x,A)") "Global s-channel mappings:"
if (allocated (forest%s_mapping)) then
    do i = 1, size (forest%s_mapping)
        if (mapping_is_s_channel (forest%s_mapping(i))) then
            write (u, "(1x,I0,':',1x)", advance="no") i
            call mapping_write (forest%s_mapping(i), unit)
        end if
    end do
else
    write (u, "(3x,A)") " [empty]"
end if
write (u, "(A)")
write (u, "(1x,A)") "Incoming particles:"
if (allocated (forest%prt_in)) then
    if (any (phs_prt_is_defined (forest%prt_in))) then
        do i = 1, size (forest%prt_in)
            if (phs_prt_is_defined (forest%prt_in(i))) then
                write (u, "(1x,A,1x,I0)") "Particle", i
                call phs_prt_write (forest%prt_in(i), u)
            end if
        end do
    else
        write (u, "(3x,A)") "[all undefined]"
    end if
else
    write (u, "(3x,A)") " [empty]"
end if
write (u, "(A)")
write (u, "(1x,A)") "Outgoing particles:"
if (allocated (forest%prt_out)) then
    if (any (phs_prt_is_defined (forest%prt_out))) then
        do i = 1, size (forest%prt_out)
            if (phs_prt_is_defined (forest%prt_out(i))) then
                write (u, "(1x,A,1x,I0)") "Particle", i
                call phs_prt_write (forest%prt_out(i), u)
            end if
        end do
    else
        write (u, "(3x,A)") "[all undefined]"
    end if
else

```

```

        write (u, "(1x,A)" " " [empty]"
    end if
    write (u, "(A)")
    write (u, "(1x,A)" "Tree particles:"
    if (allocated (forest%prt)) then
        if (any (phs_prt_is_defined (forest%prt))) then
            do i = 1, size (forest%prt)
                if (phs_prt_is_defined (forest%prt(i))) then
                    write (u, "(1x,A,1x,I0)" "Particle", i
                        call phs_prt_write (forest%prt(i), u)
                end if
            end do
        else
            write (u, "(3x,A)" "[all undefined]"
        end if
    else
        write (u, "(3x,A)" " " [empty]"
    end if
end subroutine phs_forest_write

subroutine phs_grove_write (grove, unit)
    type(phs_grove_t), intent(in) :: grove
    integer, intent(in), optional :: unit
    integer :: u
    integer :: t
    u = output_unit (unit); if (u < 0) return
    do t = 1, size (grove%tree)
        write (u, "(3x,A,I0)" "Tree      ", t
            call phs_tree_write (grove%tree(t), unit)
        end do
    write (u, "(1x,A)" "Equivalence list:"
        call equivalence_list_write (grove%equivalence_list, unit)
    end subroutine phs_grove_write

```

Deep copy.

```

<PHS forests: public>+≡
    public :: assignment(=)

<PHS forests: interfaces>+≡
    interface assignment(=)
        module procedure phs_forest_assign
    end interface

<PHS forests: procedures>+≡
    subroutine phs_forest_assign (forest_out, forest_in)
        type(phs_forest_t), intent(out) :: forest_out
        type(phs_forest_t), intent(in) :: forest_in
        forest_out%n_in = forest_in%n_in
        forest_out%n_out = forest_in%n_out
        forest_out%n_tot = forest_in%n_tot
        forest_out%n_masses = forest_in%n_masses
        forest_out%n_angles = forest_in%n_angles
        forest_out%n_dimensions = forest_in%n_dimensions
        forest_out%n_trees = forest_in%n_trees
    end subroutine

```

```

forest_out%n_equivalences = forest_in%n_equivalences
if (allocated (forest_in%flv)) then
  allocate (forest_out%flv (size (forest_in%flv)))
  forest_out%flv = forest_in%flv
end if
if (allocated (forest_in%grove)) then
  allocate (forest_out%grove (size (forest_in%grove)))
  forest_out%grove = forest_in%grove
end if
if (allocated (forest_in%grove_lookup)) then
  allocate (forest_out%grove_lookup (size (forest_in%grove_lookup)))
  forest_out%grove_lookup = forest_in%grove_lookup
end if
if (allocated (forest_in%prt_in)) then
  allocate (forest_out%prt_in (size (forest_in%prt_in)))
  forest_out%prt_in = forest_in%prt_in
end if
if (allocated (forest_in%prt_out)) then
  allocate (forest_out%prt_out (size (forest_in%prt_out)))
  forest_out%prt_out = forest_in%prt_out
end if
if (allocated (forest_in%prt)) then
  allocate (forest_out%prt (size (forest_in%prt)))
  forest_out%prt = forest_in%prt
end if
if (allocated (forest_in%s_mapping)) then
  allocate (forest_out%s_mapping (size (forest_in%s_mapping)))
  forest_out%s_mapping = forest_in%s_mapping
end if
if (allocated (forest_in%prt_combination)) then
  allocate (forest_out%prt_combination &
    (2, size (forest_in%prt_combination, 2)))
  forest_out%prt_combination = forest_in%prt_combination
end if
end subroutine phs_forest_assign

```

### 11.6.6 Accessing contents

Get the number of integration parameters

```

<PHS forests: public>+≡
  public :: phs_forest_get_n_parameters

<PHS forests: procedures>+≡
  function phs_forest_get_n_parameters (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_dimensions
  end function phs_forest_get_n_parameters

```

Get the number of integration channels

```

<PHS forests: public>+≡
  public :: phs_forest_get_n_channels

```

```

<PHS forests: procedures>+≡
  function phs_forest_get_n_channels (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_trees
  end function phs_forest_get_n_channels

```

Get the number of groves

```

<PHS forests: public>+≡
  public :: phs_forest_get_n_groves

<PHS forests: procedures>+≡
  function phs_forest_get_n_groves (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = size (forest%grove)
  end function phs_forest_get_n_groves

```

Get the index bounds for a specific grove.

```

<PHS forests: public>+≡
  public :: phs_forest_get_grove_bounds

<PHS forests: procedures>+≡
  subroutine phs_forest_get_grove_bounds (forest, g, i0, i1, n)
    type(phs_forest_t), intent(in) :: forest
    integer, intent(in) :: g
    integer, intent(out) :: i0, i1, n
    n = size (forest%grove(g)%tree)
    i0 = forest%grove(g)%tree_count_offset + 1
    i1 = forest%grove(g)%tree_count_offset + n
  end subroutine phs_forest_get_grove_bounds

```

Get the number of equivalences

```

<PHS forests: public>+≡
  public :: phs_forest_get_n_equivalences

<PHS forests: procedures>+≡
  function phs_forest_get_n_equivalences (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_equivalences
  end function phs_forest_get_n_equivalences

```

Return true if a particular channel has a global (s-channel) mapping; also return the resonance mass and width for this mapping.

```

<PHS forests: public>+≡
  public :: phs_forest_get_s_mapping

<PHS forests: procedures>+≡
  subroutine phs_forest_get_s_mapping (forest, channel, flag, mass, width)
    type(phs_forest_t), intent(in) :: forest
    integer, intent(in) :: channel
    logical, intent(out) :: flag

```

```

real(default), intent(out) :: mass, width
flag = mapping_is_s_channel (forest%s_mapping(channel))
if (flag) then
    mass = mapping_get_mass (forest%s_mapping(channel))
    width = mapping_get_width (forest%s_mapping(channel))
else
    mass = 0
    width = 0
end if
end subroutine phs_forest_get_s_mapping

```

### 11.6.7 Read the phase space setup from file

The phase space setup is stored in a file. The file may be generated by the `cascades` module below, or by other means. This file has to be read and parsed to create the PHS forest as the internal phase-space representation.

Create lexer and syntax:

```

<PHS forests: procedures>+≡
subroutine define_phs_forest_syntax (ifile)
    type(ifile_t) :: ifile
    call ifile_append (ifile, "SEQ phase_space_list = process_phase_space*")
    call ifile_append (ifile, "SEQ process_phase_space = " &
        // "process_def process_header phase_space")
    call ifile_append (ifile, "SEQ process_def = process process_list")
    call ifile_append (ifile, "KEY process")
    call ifile_append (ifile, "LIS process_list = process_tag*")
    call ifile_append (ifile, "IDE process_tag")
    call ifile_append (ifile, "SEQ process_header = " &
        // "md5sum_process = md5sum " &
        // "md5sum_model_par = md5sum " &
        // "md5sum_phs_config = md5sum " &
        // "sqrts = real " &
        // "m_threshold_s = real " &
        // "m_threshold_t = real " &
        // "off_shell = integer " &
        // "t_channel = integer " &
        // "keep_nonresonant = logical")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "KEY md5sum_process")
    call ifile_append (ifile, "KEY md5sum_model_par")
    call ifile_append (ifile, "KEY md5sum_phs_config")
    call ifile_append (ifile, "KEY sqrts")
    call ifile_append (ifile, "KEY m_threshold_s")
    call ifile_append (ifile, "KEY m_threshold_t")
    call ifile_append (ifile, "KEY off_shell")
    call ifile_append (ifile, "KEY t_channel")
    call ifile_append (ifile, "KEY keep_nonresonant")
    call ifile_append (ifile, "QUO md5sum = '""' ... '""'")
    call ifile_append (ifile, "REA real")
    call ifile_append (ifile, "INT integer")
    call ifile_append (ifile, "IDE logical")
    call ifile_append (ifile, "SEQ phase_space = grove_def+")

```



```

call ifile_append (ifile, "SEQ grove_def = grove tree_def+")
call ifile_append (ifile, "KEY grove")
call ifile_append (ifile, "SEQ tree_def = tree bincodes mapping*")
call ifile_append (ifile, "KEY tree")
call ifile_append (ifile, "SEQ bincodes = bincodes+")
call ifile_append (ifile, "INT bincodes")
call ifile_append (ifile, "SEQ mapping = map bincodes channel pdg")
call ifile_append (ifile, "KEY map")
call ifile_append (ifile, "ALT channel = s_channel | t_channel | u_channel | collinear | infrared")
call ifile_append (ifile, "KEY s_channel")
! call ifile_append (ifile, "KEY t_channel")
call ifile_append (ifile, "KEY u_channel")
call ifile_append (ifile, "KEY collinear")
call ifile_append (ifile, "KEY infrared")
call ifile_append (ifile, "KEY radiation")
call ifile_append (ifile, "INT pdg")
end subroutine define_phs_forest_syntax

```

The model-file syntax and lexer are fixed, therefore stored as module variables:

```

<PHS forests: variables>≡
type(syntax_t), target, save :: syntax_phs_forest

<PHS forests: public>+≡
public :: syntax_phs_forest_init

<PHS forests: procedures>+≡
subroutine syntax_phs_forest_init ()
type(ifile_t) :: ifile
call define_phs_forest_syntax (ifile)
call syntax_init (syntax_phs_forest, ifile)
call ifile_final (ifile)
end subroutine syntax_phs_forest_init

<PHS forests: procedures>+≡
subroutine lexer_init_phs_forest (lexer)
type(lexer_t), intent(out) :: lexer
call lexer_init (lexer, &
comment_chars = "#!", &
quote_chars = "'", &
quote_match = '"', &
single_chars = "\"", &
special_class = (/ "=", /) , &
keyword_list = syntax_get_keyword_list_ptr (syntax_phs_forest))
end subroutine lexer_init_phs_forest

<PHS forests: public>+≡
public :: syntax_phs_forest_final

<PHS forests: procedures>+≡
subroutine syntax_phs_forest_final ()
call syntax_final (syntax_phs_forest)
end subroutine syntax_phs_forest_final

```

```

(PHS forests: public)+≡
  public :: syntax_phs_forest_write

(PHS forests: procedures)+≡
  subroutine syntax_phs_forest_write (unit)
    integer, intent(in), optional :: unit
    call syntax_write (syntax_phs_forest, unit)
  end subroutine syntax_phs_forest_write

```

The concrete parser and interpreter. Generate an input stream for the external `unit`, read the parse tree (with given `syntax` and `lexer`) from this stream, and transfer the contents of the parse tree to the PHS forest.

We look for the matching `process` tag, count groves and trees for initializing the forest, and fill the trees.

If the optional parameters are set, compare the parameters stored in the file to those. Set `match` true if everything agrees.

```

(PHS forests: public)+≡
  public :: phs_forest_read

(PHS forests: interfaces)+≡
  interface phs_forest_read
    module procedure phs_forest_read_file
    module procedure phs_forest_read_unit
    module procedure phs_forest_read_parse_tree
  end interface

(PHS forests: procedures)+≡
  subroutine phs_forest_read_file &
    (forest, filename, process_id, n_in, n_out, model, found, &
     md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    type(phs_forest_t), intent(out) :: forest
    type(string_t), intent(in) :: filename
    type(string_t), intent(in) :: process_id
    integer, intent(in) :: n_in, n_out
    type(model_t), intent(in), target :: model
    logical, intent(out) :: found
    character(32), intent(in), optional :: &
      md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out), optional :: match
    type(parse_tree_t), target :: parse_tree
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_phs_forest (lexer)
    call stream_init (stream, char (filename))
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
    call phs_forest_read (forest, parse_tree, &
      process_id, n_in, n_out, model, found, &
      md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    call stream_final (stream)
    call lexer_final (lexer)
    call parse_tree_final (parse_tree)
  end subroutine phs_forest_read_file

```

```

subroutine phs_forest_read_unit &
    (forest, unit, process_id, n_in, n_out, model, found, &
     md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    type(phs_forest_t), intent(out) :: forest
    integer, intent(in) :: unit
    type(string_t), intent(in) :: process_id
    integer, intent(in) :: n_in, n_out
    type(model_t), intent(in), target :: model
    logical, intent(out) :: found
    character(32), intent(in), optional :: &
        md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out), optional :: match
    type(parse_tree_t), target :: parse_tree
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_phs_forest (lexer)
    call stream_init (stream, unit)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
    call phs_forest_read (forest, parse_tree, &
        process_id, n_in, n_out, model, found, &
        md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    call stream_final (stream)
    call lexer_final (lexer)
    call parse_tree_final (parse_tree)
end subroutine phs_forest_read_unit

subroutine phs_forest_read_parse_tree &
    (forest, parse_tree, process_id, n_in, n_out, model, found, &
     md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    type(phs_forest_t), intent(out) :: forest
    type(parse_tree_t), intent(in), target :: parse_tree
    type(string_t), intent(in) :: process_id
    integer, intent(in) :: n_in, n_out
    type(model_t), intent(in), target :: model
    logical, intent(out) :: found
    character(32), intent(in), optional :: &
        md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out), optional :: match
    type(parse_node_t), pointer :: node_header, node_phs, node_grove
    integer :: n_grove, g
    integer, dimension(:), allocatable :: n_tree
    integer :: t
!    call parse_tree_write (parse_tree)
    node_header => parse_tree_get_process_ptr (parse_tree, process_id)
    found = associated (node_header); if (.not. found) return
    if (present (match)) then
        call phs_forest_check_input (node_header, &
            md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
        if (.not. match) return
    end if

```

```

node_phs => parse_node_get_next_ptr (node_header)
n_grove = parse_node_get_n_sub (node_phs)
allocate (n_tree (n_grove))
do g = 1, n_grove
    node_grove => parse_node_get_sub_ptr (node_phs, g)
    n_tree(g) = parse_node_get_n_sub (node_grove) - 1
end do
call phs_forest_init (forest, n_tree, n_in, n_out)
do g = 1, n_grove
    node_grove => parse_node_get_sub_ptr (node_phs, g)
    do t = 1, n_tree(g)
        call phs_tree_set (forest%grove(g)%tree(t), &
            parse_node_get_sub_ptr (node_grove, t+1), model)
    end do
end do
end subroutine phs_forest_read_parse_tree

```

Check the input for consistency. If any MD5 sum or phase-space parameter disagrees, the phase-space file cannot be used. The MD5 sum checks are skipped if the stored MD5 sum is empty.

(*PHS forests: procedures*) +=

```

subroutine phs_forest_check_input (pn_header, &
    md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    type(parse_node_t), intent(in), target :: pn_header
    character(32), intent(in) :: &
        md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out) :: match
    type(parse_node_t), pointer :: pn_md5sum, pn_rval, pn_lval, pn_lval
    character(32) :: md5sum
    type(phs_parameters_t) :: phs_par_old
    character(1) :: lstr
    pn_md5sum => parse_node_get_sub_ptr (pn_header, 3)
    md5sum = parse_node_get_string (pn_md5sum)
    if (md5sum /= "" .and. md5sum /= md5sum_process) then
        call msg_message ("Phase space: discarding old configuration &
            &(process changed)")
        match = .false.; return
    end if
    pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
    md5sum = parse_node_get_string (pn_md5sum)
    if (md5sum /= "" .and. md5sum /= md5sum_model_par) then
        call msg_message ("Phase space: discarding old configuration &
            &(model parameters changed)")
        match = .false.; return
    end if
    pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
    md5sum = parse_node_get_string (pn_md5sum)
    if (md5sum /= "" .and. md5sum /= md5sum_phs_config) then
        call msg_message ("Phase space: discarding old configuration &
            &(configuration parameters changed)")
        match = .false.; return
    end if
end if

```

```

if (present (phs_par)) then
  pn_rval => parse_node_get_next_ptr (pn_md5sum, 3)
  phs_par_old%sqrts = parse_node_get_real (pn_rval)
  pn_rval => parse_node_get_next_ptr (pn_rval, 3)
  phs_par_old%m_threshold_s = parse_node_get_real (pn_rval)
  pn_rval => parse_node_get_next_ptr (pn_rval, 3)
  phs_par_old%m_threshold_t = parse_node_get_real (pn_rval)
  pn_ival => parse_node_get_next_ptr (pn_rval, 3)
  phs_par_old%off_shell = parse_node_get_integer (pn_ival)
  pn_ival => parse_node_get_next_ptr (pn_ival, 3)
  phs_par_old%t_channel = parse_node_get_integer (pn_ival)
  pn_lval => parse_node_get_next_ptr (pn_ival, 3)
  lstr = parse_node_get_string (pn_lval)
  read (lstr, "(L1)") phs_par_old%keep_nonresonant
  if (phs_par_old /= phs_par) then
    call msg_message &
      ("Phase space: discarding old configuration &
      &(configuration parameters changed)")
    match = .false.; return
  end if
end if
match = .true.
end subroutine phs_forest_check_input

```

Initialize a specific tree in the forest, using the contents of the 'tree' node. First, count the bincodes, allocate an array and read them in, and make the tree. Each *t*-channel tree is flipped to *s*-channel. Then, find mappings and initialize them.

(*PHS forests: procedures*) +=

```

subroutine phs_tree_set (tree, node, model)
  type(phs_tree_t), intent(inout) :: tree
  type(parse_node_t), intent(in), target :: node
  type(model_t), intent(in), target :: model
  type(parse_node_t), pointer :: node_bincodes, node_mapping
  integer :: n_bincodes
  integer(TC), dimension(:), allocatable :: bincode
  integer :: b, n_mappings, m
  integer(TC) :: k
  type(string_t) :: type
  integer :: pdg
  node_bincodes => parse_node_get_sub_ptr (node, 2)
  n_bincodes = parse_node_get_n_sub (node_bincodes)
  allocate (bincode (n_bincodes))
  do b = 1, n_bincodes
    bincode(b) = parse_node_get_integer &
      (parse_node_get_sub_ptr (node_bincodes, b))
  end do
  call phs_tree_from_array (tree, bincode)
  call phs_tree_flip_t_to_s_channel (tree)
  call phs_tree_canonicalize (tree)
  n_mappings = parse_node_get_n_sub (node) - 2
  do m = 1, n_mappings
    node_mapping => parse_node_get_sub_ptr (node, m+2)
    k = parse_node_get_integer &

```

```

        (parse_node_get_sub_ptr (node_mapping, 2))
    type = parse_node_get_key &
        (parse_node_get_sub_ptr (node_mapping, 3))
    pdg = parse_node_get_integer &
        (parse_node_get_sub_ptr (node_mapping, 4))
    call phs_tree_init_mapping (tree, k, type, pdg, model)
end do
end subroutine phs_tree_set

```

### 11.6.8 Preparation

The trees that we read from file do not carry flavor information. This is set separately:

The flavor list must be unique for a unique set of masses; if a given particle can have different flavor, the mass must be degenerate, so we can choose one of the possible flavor combinations.

```

<PHS forests: public>+≡
    public :: phs_forest_set_flavors

<PHS forests: procedures>+≡
    subroutine phs_forest_set_flavors (forest, flv)
        type(phs_forest_t), intent(inout) :: forest
        type(flavor_t), dimension(:), intent(in) :: flv
        allocate (forest%flv (size (flv)))
        forest%flv = flv
    end subroutine phs_forest_set_flavors

```

Once the parameter set is fixed, the masses and the widths of the particles are known and the `mass_sum` arrays as well as the mapping parameters can be computed. Note that order is important: we first compute the mass sums, then the ordinary mappings. The resonances obtained here determine the effective masses, which in turn are used to implement step mappings for resonance decay products that are not mapped otherwise.

```

<PHS forests: public>+≡
    public :: phs_forest_set_parameters

<PHS forests: procedures>+≡
    subroutine phs_forest_set_parameters &
        (forest, mapping_defaults, variable_limits)
        type(phs_forest_t), intent(inout) :: forest
        type(mapping_defaults_t), intent(in) :: mapping_defaults
        logical, intent(in) :: variable_limits
        integer :: g, t
        do g = 1, size (forest%grove)
            do t = 1, size (forest%grove(g)%tree)
                call phs_tree_set_mass_sum &
                    (forest%grove(g)%tree(t), forest%flv(forest%n_in+1:))
                call phs_tree_set_mapping_parameters (forest%grove(g)%tree(t), &
                    mapping_defaults, variable_limits)
                call phs_tree_set_effective_masses (forest%grove(g)%tree(t))
                if (mapping_defaults%step_mapping) then
                    call phs_tree_set_step_mappings (forest%grove(g)%tree(t), &

```

```

                                mapping_defaults%step_mapping_exp, variable_limits)
        end if
    end do
end do
end subroutine phs_forest_set_parameters

```

Generate the particle combination table. Scan all trees and merge their individual combination tables. At the end, valid entries are non-zero, and they indicate the indices of a pair of particles to be combined to a new particle. If a particle is accessible by more than one tree (this is usual), only keep the first possibility.

```

<PHS forests: public>+≡
    public :: phs_forest_setup_prt_combinations

<PHS forests: procedures>+≡
    subroutine phs_forest_setup_prt_combinations (forest)
        type(phs_forest_t), intent(inout) :: forest
        integer :: g, t
        integer, dimension(:,,:), allocatable :: tree_prt_combination
        forest%prt_combination = 0
        allocate (tree_prt_combination (2, size (forest%prt_combination, 2)))
        do g = 1, size (forest%grove)
            do t = 1, size (forest%grove(g)%tree)
                call phs_tree_setup_prt_combinations &
                    (forest%grove(g)%tree(t), tree_prt_combination)
                where (tree_prt_combination /= 0 .and. forest%prt_combination == 0)
                    forest%prt_combination = tree_prt_combination
                end where
            end do
        end do
    end subroutine phs_forest_setup_prt_combinations

```

### 11.6.9 Accessing the particle arrays

Set the incoming particles from the contents of an interaction.

```

<PHS forests: public>+≡
    public :: phs_forest_set_prt_in

<PHS forests: interfaces>+≡
    interface phs_forest_set_prt_in
        module procedure phs_forest_set_prt_in_int, phs_forest_set_prt_in_mom
    end interface phs_forest_set_prt_in

<PHS forests: procedures>+≡
    subroutine phs_forest_set_prt_in_int (forest, int, lt_cm_to_lab)
        type(phs_forest_t), intent(inout) :: forest
        type(interaction_t), intent(in) :: int
        type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
        if (present (lt_cm_to_lab)) then
            call phs_prt_set_momentum (forest%prt_in, &
                inverse (lt_cm_to_lab) * &
                interaction_get_momenta (int, outgoing=.false.))
        else
            call phs_prt_set_momentum (forest%prt_in, &

```

```

        interaction_get_momenta (int, outgoing=.false.))
    end if
    call phs_prt_set_msq (forest%prt_in, &
        flavor_get_mass (forest%flv(:forest%n_in)) ** 2)
    call phs_prt_set_defined (forest%prt_in)
end subroutine phs_forest_set_prt_in_int

subroutine phs_forest_set_prt_in_mom (forest, mom, lt_cm_to_lab)
    type(phs_forest_t), intent(inout) :: forest
    type(vector4_t), dimension(size (forest%prt_in)), intent(in) :: mom
    type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
    if (present (lt_cm_to_lab)) then
        call phs_prt_set_momentum (forest%prt_in, &
            inverse (lt_cm_to_lab) * mom)
    else
        call phs_prt_set_momentum (forest%prt_in, mom)
    end if
    call phs_prt_set_msq (forest%prt_in, &
        flavor_get_mass (forest%flv(:forest%n_in)) ** 2)
    call phs_prt_set_defined (forest%prt_in)
end subroutine phs_forest_set_prt_in_mom

```

Set the outgoing particles from the contents of an interaction.

```

<PHS forests: public>+≡
    public :: phs_forest_set_prt_out

<PHS forests: interfaces>+≡
    interface phs_forest_set_prt_out
        module procedure phs_forest_set_prt_out_int, phs_forest_set_prt_out_mom
    end interface phs_forest_set_prt_out

<PHS forests: procedures>+≡
    subroutine phs_forest_set_prt_out_int (forest, int, lt_cm_to_lab)
        type(phs_forest_t), intent(inout) :: forest
        type(interaction_t), intent(in) :: int
        type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
        if (present (lt_cm_to_lab)) then
            call phs_prt_set_momentum (forest%prt_out, &
                inverse (lt_cm_to_lab) * &
                interaction_get_momenta (int, outgoing=.true.))
        else
            call phs_prt_set_momentum (forest%prt_out, &
                interaction_get_momenta (int, outgoing=.true.))
        end if
        call phs_prt_set_msq (forest%prt_out, &
            flavor_get_mass (forest%flv(forest%n_in+1:)) ** 2)
        call phs_prt_set_defined (forest%prt_out)
    end subroutine phs_forest_set_prt_out_int

    subroutine phs_forest_set_prt_out_mom (forest, mom, lt_cm_to_lab)
        type(phs_forest_t), intent(inout) :: forest
        type(vector4_t), dimension(size (forest%prt_out)), intent(in) :: mom
        type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
        if (present (lt_cm_to_lab)) then
            call phs_prt_set_momentum (forest%prt_out, &

```



```

        inverse (lt_cm_to_lab) * mom)
    else
        call phs_prt_set_momentum (forest%prt_out, mom)
    end if
    call phs_prt_set_msq (forest%prt_out, &
        flavor_get_mass (forest%flv(forest%n_in+1:)) ** 2)
    call phs_prt_set_defined (forest%prt_out)
end subroutine phs_forest_set_prt_out_mom

```

Combine particles as described by the particle combination table. Particle momentum sums will be calculated only if the resulting particle is contained in at least one of the trees in the current forest. The others are kept undefined.

```

<PHS forests: public>+≡
    public :: phs_forest_combine_particles

<PHS forests: procedures>+≡
    subroutine phs_forest_combine_particles (forest)
        type(phs_forest_t), intent(inout) :: forest
        integer :: k
        integer, dimension(2) :: kk
        do k = 1, size (forest%prt_combination, 2)
            kk = forest%prt_combination(:,k)
            if (kk(1) /= 0) then
                call phs_prt_combine (forest%prt(k), &
                    forest%prt(kk(1)), forest%prt(kk(2)))
            end if
        end do
    end subroutine phs_forest_combine_particles

```

Extract the outgoing particles and insert into an interaction.

```

<PHS forests: public>+≡
    public :: phs_forest_get_prt_out

<PHS forests: procedures>+≡
    subroutine phs_forest_get_prt_out (forest, int, lt_cm_to_lab)
        type(phs_forest_t), intent(in) :: forest
        type(interaction_t), intent(inout) :: int
        type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
        if (present (lt_cm_to_lab)) then
            call interaction_set_momenta (int, &
                lt_cm_to_lab * &
                phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
        else
            call interaction_set_momenta (int, &
                phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
        end if
    end subroutine phs_forest_get_prt_out

```

Extract the outgoing particle momenta

```

<PHS forests: public>+≡
    public :: phs_forest_get_momenta_out

```

*(PHS forests: procedures)+≡*

```
function phs_forest_get_momenta_out (forest, lt_cm_to_lab) result (p)
  type(phs_forest_t), intent(in) :: forest
  type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
  type(vector4_t), dimension(size (forest%prt_out)) :: p
  p = phs_prt_get_momentum (forest%prt_out)
  if (present (lt_cm_to_lab)) p = p * lt_cm_to_lab
end function phs_forest_get_momenta_out
```

### 11.6.10 Find equivalences among phase-space trees

Scan phase space for equivalences. We generate the complete set of unique permutations for the given list of outgoing particles, and use this for scanning equivalences within each grove. We scan all pairs of trees, using all permutations. This implies that trivial equivalences are included, and equivalences between different trees are recorded twice. This is intentional.

*(PHS forests: procedures)+≡*

```
subroutine phs_grove_set_equivalences (grove, perm_array)
  type(phs_grove_t), intent(inout) :: grove
  type(permutation_t), dimension(:), intent(in) :: perm_array
  type(equivalence_t), pointer :: eq
  integer :: t1, t2, i
  do t1 = 1, size (grove%tree)
    do t2 = 1, size (grove%tree)
      SCAN_PERM: do i = 1, size (perm_array)
        if (phs_tree_equivalent &
            (grove%tree(t1), grove%tree(t2), perm_array(i))) then
          call equivalence_list_add &
            (grove%equivalence_list, t1, t2, perm_array(i))
          eq => grove%equivalence_list%last
          call phs_tree_find_msq_permutation &
            (grove%tree(t1), grove%tree(t2), eq%perm, &
             eq%msq_perm)
          call phs_tree_find_angle_permutation &
            (grove%tree(t1), grove%tree(t2), eq%perm, &
             eq%angle_perm, eq%angle_sig)
        end if
      end do SCAN_PERM
    end do
  end do
end subroutine phs_grove_set_equivalences
```

*(PHS forests: public)+≡*

```
public :: phs_forest_set_equivalences
```

*(PHS forests: procedures)+≡*

```
subroutine phs_forest_set_equivalences (forest)
  type(phs_forest_t), intent(inout) :: forest
  type(permutation_t), dimension(:), allocatable :: perm_array
  integer :: i
  call permutation_array_make &
    (perm_array, flavor_get_pdg (forest%flv(forest%n_in+1:)))
  do i = 1, size (forest%grove)
```

```

        call phs_grove_set_equivalences (forest%grove(i), perm_array)
    end do
    forest%n_equivalences = sum (forest%grove%equivalence_list%length)
end subroutine phs_forest_set_equivalences

```

### 11.6.11 Interface for channel equivalences

Here, we store the equivalence list in the appropriate containers that the `phs_base` module provides. There is one separate list for each channel.

```

<PHS forests: public>+=
    public :: phs_forest_get_equivalences

<PHS forests: procedures>+=
    subroutine phs_forest_get_equivalences (forest, channel, azimuthal_dependence)
        type(phs_forest_t), intent(in) :: forest
        type(phs_channel_t), dimension(:), intent(out) :: channel
        logical, intent(in) :: azimuthal_dependence
        integer :: n_masses, n_angles
        integer :: mode_azimuthal_angle
        integer, dimension(:), allocatable :: n_eq
        type(equivalence_t), pointer :: eq
        integer, dimension(:), allocatable :: perm, mode
        integer :: g, c, j, left, right
        n_masses = forest%n_masses
        n_angles = forest%n_angles
        allocate (n_eq (forest%n_trees), source = 0)
        allocate (perm (forest%n_dimensions))
        allocate (mode (forest%n_dimensions), source = EQ_IDENTITY)
        do g = 1, size (forest%grove)
            eq => forest%grove(g)%equivalence_list%first
            do while (associated (eq))
                left = eq%left + forest%grove(g)%tree_count_offset
                n_eq(left) = n_eq(left) + 1
                eq => eq%next
            end do
        end do
        do c = 1, size (channel)
            allocate (channel(c)%eq (n_eq(c)))
            do j = 1, n_eq(c)
                call channel(c)%eq(j)%init (forest%n_dimensions)
            end do
        end do
        n_eq = 0
        if (azimuthal_dependence) then
            mode_azimuthal_angle = EQ_IDENTITY
        else
            mode_azimuthal_angle = EQ_INVARIANT
        end if
        do g = 1, size (forest%grove)
            eq => forest%grove(g)%equivalence_list%first
            do while (associated (eq))
                left = eq%left + forest%grove(g)%tree_count_offset
                right = eq%right + forest%grove(g)%tree_count_offset

```

```

do j = 1, n_masses
  perm(j) = permute (j, eq%msq_perm)
  mode(j) = EQ_IDENTITY
end do
do j = 1, n_angles
  perm(n_masses+j) = n_masses + permute (j, eq%angle_perm)
  if (j == 1) then
    mode(n_masses+j) = mode_azimuthal_angle ! first az. angle
  else if (mod(j,2) == 1) then
    mode(n_masses+j) = EQ_SYMMETRIC ! other az. angles
  else if (eq%angle_sig(j)) then
    mode(n_masses+j) = EQ_IDENTITY ! polar angle +
  else
    mode(n_masses+j) = EQ_INVERT ! polar angle -
  end if
end do
n_eq(left) = n_eq(left) + 1
associate (eq_cur => channel(left)%eq(n_eq(left)))
  eq_cur%c = right
  eq_cur%perm = perm
  eq_cur%mode = mode
end associate
eq => eq%next
end do
end do
end subroutine phs_forest_get_equivalences

```

Obsolete:

Transform the equivalence lists into a `vamp_equivalence_list` object. The additional information that we need is: the number of extra integration dimensions (associated to structure functions), which ones of those correspond to external event generators (so that the binning must not be adapted), and whether there is any dependence on the first azimuthal angle (so its binning may be adapted or fixed).

The permutations of masses and angles are translated into permutations of integration dimensions, where the correct mapping modes are important: `msq` values are always mapped 1:1 while azimuthal angles may need an offset by 1/2 and polar angles may need an inversion. The first azimuthal angle should not be adapted to since the dependence of the matrix element is (usually) trivial.

```

<XXX PHS forests: public>≡
  public :: phs_forest_setup_vamp_equivalences

<XXX PHS forests: procedures>≡
  subroutine phs_forest_setup_vamp_equivalences &
    (forest, n_dim_extra, externally_generated, azimuthal_dependence, &
     vamp_eq)
    type(phs_forest_t), intent(in) :: forest
    integer, intent(in) :: n_dim_extra
    logical, dimension(n_dim_extra), intent(in) :: externally_generated
    logical, intent(in) :: azimuthal_dependence
    type(vamp_equivalences_t), intent(out) :: vamp_eq
    integer :: n_equivalences, n_channels, n_dim, n_masses, n_angles
    integer, dimension(forest%n_dimensions + n_dim_extra) :: perm, mode

```

```

integer :: mode_azimuthal_angle
type(equivalence_t), pointer :: eq
integer :: i, j, g
integer :: left, right
n_equivalences = forest%n_equivalences
n_channels = forest%n_trees
n_dim = forest%n_dimensions
n_masses = forest%n_masses
n_angles = forest%n_angles
call vamp_equivalences_init &
    (vamp_eq, n_equivalences, n_channels, n_dim + n_dim_extra)
if (azimuthal_dependence) then
    mode_azimuthal_angle = VEQ_IDENTITY
else
    mode_azimuthal_angle = VEQ_INVARIANT
end if
g = 0
eq => null ()
do i = 1, n_equivalences
    if (.not. associated (eq)) then
        g = g + 1
        eq => forest%grove(g)%equivalence_list%first
    end if
    do j = 1, n_masses
        perm(j) = permute (j, eq%msq_perm)
        mode(j) = VEQ_IDENTITY
    end do
    do j = 1, n_angles
        perm(n_masses+j) = n_masses + permute (j, eq%angle_perm)
        if (j == 1) then
            mode(n_masses+j) = mode_azimuthal_angle    ! first azimuthal angle
        else if (mod(j,2) == 1) then
            mode(n_masses+j) = VEQ_SYMMETRIC            ! other azimuthal angles
        else if (eq%angle_sig(j)) then
            mode(n_masses+j) = VEQ_IDENTITY             ! polar angle +
        else
            mode(n_masses+j) = VEQ_INVERT              ! polar angle -
        end if
    end do
    do j = 1, n_dim_extra
        perm(n_dim+j) = n_dim + j
        if (externally_generated(j)) then
            mode(n_dim+j) = VEQ_INVARIANT
        else
            mode(n_dim+j) = VEQ_IDENTITY
        end if
    end do
    left = eq%left + forest%grove(g)%tree_count_offset
    right = eq%right + forest%grove(g)%tree_count_offset
    call vamp_equivalence_set (vamp_eq, i, left, right, perm, mode)
    eq => eq%next
end do
call vamp_equivalences_complete (vamp_eq)
end subroutine phs_forest_setup_vamp_equivalences

```

### 11.6.12 Phase-space evaluation

Given one row of the `x` parameter array and the corresponding channel index, compute first all relevant momenta and then recover the remainder of the `x` array, the Jacobians `phs_factor`, and the phase-space volume.

The output argument `ok` indicates whether this was successful.

*(PHS forests: public)*+≡

```
public :: phs_forest_evaluate_phase_space
```

*(PHS forests: procedures)*+≡

```
subroutine phs_forest_evaluate_phase_space &
  (forest, channel, active, sqrts, x, phs_factor, volume, ok)
  type(phs_forest_t), intent(inout) :: forest
  integer, intent(in) :: channel
  logical, dimension(:), intent(in) :: active
  real(default), intent(in) :: sqrts
  real(default), dimension(:,:), intent(inout) :: x
  real(default), dimension(:), intent(out) :: phs_factor
  real(default), intent(out) :: volume
  logical, intent(out) :: ok
  integer :: g, t, ch, s
  integer(TC) :: k, k_root, k_in

  g = forest%grove_lookup (channel)
  t = channel - forest%grove(g)%tree_count_offset
  call phs_prt_set_undefined (forest%prt)
  call phs_prt_set_undefined (forest%prt_out)
  k_in = forest%n_tot

  do k = 1, forest%n_in
    forest%prt(ibset(0, k_in-k)) = forest%prt_in(k)
  end do

  do k = 1, forest%n_out
    call phs_prt_set_msq (forest%prt(ibset(0, k-1)), &
      flavor_get_mass (forest%flv(forest%n_in+k)) ** 2)
  end do

  k_root = 2**forest%n_out - 1
  select case (forest%n_in)
  case (1)
    forest%prt(k_root) = forest%prt_in(1)
  case (2)
    call phs_prt_combine &
      (forest%prt(k_root), forest%prt_in(1), forest%prt_in(2))
  end select
  call phs_tree_compute_momenta_from_x (forest%grove(g)%tree(t), &
    forest%prt, phs_factor(channel), volume, sqrts, x(:, channel), ok)
  if (ok) then
    !$OMP PARALLEL PRIVATE (g,t,ch,s) SHARED(forest,sqrt,x,channel)
```

```

!$OMP DO SCHEDULE(STATIC)
do ch = 1, forest%n_trees
  if (ch == channel) cycle
  if (active(ch)) then
    s = ch
    g = 1
    do while (s > size (forest%grove(g)%tree))
      s = s - size(forest%grove(g)%tree)
      g = g+1
    end do
    t = s

    call phs_tree_combine_particles &
      (forest%grove(g)%tree(t), forest%prt)
    call phs_tree_compute_x_from_momenta &
      (forest%grove(g)%tree(t), &
       forest%prt, phs_factor(ch), sqrts, x(:,ch))
  end if
end do
!$OMP END DO

!$OMP DO
do k = 1, forest%n_out
  forest%prt_out(k) = forest%prt(ibset(0,k-1))
end do
!$OMP END DO
!$OMP END PARALLEL
end if
end subroutine phs_forest_evaluate_phase_space

```

The first part of the previous routine, without recovering the  $x$  values. Here the PHS factor is `intent(inout)` because only one of the channels is defined.

```

<PHS forests: public>+≡
  public :: phs_forest_evaluate_momenta

<PHS forests: procedures>+≡
  subroutine phs_forest_evaluate_momenta &
    (forest, channel, active, sqrts, x, phs_factor, volume, ok)
    type(phs_forest_t), intent(inout) :: forest
    integer, intent(in) :: channel
    logical, dimension(:), intent(in) :: active
    real(default), intent(in) :: sqrts
    real(default), dimension(:, :), intent(in) :: x
    real(default), dimension(:), intent(inout) :: phs_factor
    real(default), intent(out) :: volume
    logical, intent(out) :: ok
    integer :: g, t, ch, s
    integer(TC) :: k, k_root, k_in
    g = forest%grove_lookup (channel)
    t = channel - forest%grove(g)%tree_count_offset
    call phs_prt_set_undefined (forest%prt)
    call phs_prt_set_undefined (forest%prt_out)
    k_in = forest%n_tot
    forall (k = 1:forest%n_in)

```

```

        forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
    end forall
    do k = 1, forest%n_out
        call phs_prt_set_msq (forest%prt(ibset(0,k-1)), &
            flavor_get_mass (forest%flv(forest%n_in+k)) ** 2)
    end do
    k_root = 2**forest%n_out - 1
    select case (forest%n_in)
    case (1)
        forest%prt(k_root) = forest%prt_in(1)
    case (2)
        call phs_prt_combine &
            (forest%prt(k_root), forest%prt_in(1), forest%prt_in(2))
    end select
    call phs_tree_compute_momenta_from_x (forest%grove(g)%tree(t), &
        forest%prt, phs_factor(channel), volume, sqrts, x(:,channel), ok)
    if (ok) then
        call phs_forest_combine_particles (forest)
        do k = 1, forest%n_out
            forest%prt_out(k) = forest%prt(ibset(0,k-1))
        end do
    end if
end subroutine phs_forest_evaluate_momenta

```

The remainder: recover  $x$  values for all channels except for the current channel.

*(PHS forests: public)*+≡

```
public :: phs_forest_evaluate_other_channels
```

*(PHS forests: procedures)*+≡

```

subroutine phs_forest_evaluate_other_channels &
    (forest, channel, active, sqrts, x, phs_factor)
    type(phs_forest_t), intent(inout) :: forest
    integer, intent(in) :: channel
    logical, dimension(:), intent(in) :: active
    real(default), intent(in) :: sqrts
    real(default), dimension(:,:), intent(inout) :: x
    real(default), dimension(:), intent(inout) :: phs_factor
    integer :: g, t, ch, w, k, s
    integer :: OMP_GET_NUM_THREADS

    w = 0
    !$OMP PARALLEL PRIVATE (g,t,ch,s,k) SHARED(forest,sqrt,x,channel,w)

    !$OMP DO REDUCTION(+:w)
    do g = 1, size (forest%grove)
        do t = 1, size (forest%grove(g)%tree)
            w = w+1
        end do
    end do
    !$OMP END DO

    !$OMP DO SCHEDULE(STATIC)
    do ch = 1, w
        if (ch == channel) cycle
    end do
    !$OMP END DO

```



```

        if (active(ch)) then
            s = ch
            g = 1
            t = 1
            do while (s > size (forest%grove(g)%tree))
                s = s - size(forest%grove(g)%tree)
                g = g+1
            end do
            t = s
            call phs_tree_compute_x_from_momenta &
                (forest%grove(g)%tree(t), &
                 forest%prt, phs_factor(ch), sqrts, x(:,ch))
        end if
    end do
!$OMP END DO
!$OMP END PARALLEL

end subroutine phs_forest_evaluate_other_channels

```

The complement: recover one row of the  $x$  array and the associated Jacobian entry, corresponding to **channel**, from incoming and outgoing momenta. Also compute the phase-space volume.

```

<PHS forests: public>+≡
    public :: phs_forest_recover_channel

<PHS forests: procedures>+≡
    subroutine phs_forest_recover_channel &
        (forest, channel, sqrts, x, phs_factor, volume)
        type(phs_forest_t), intent(inout) :: forest
        integer, intent(in) :: channel
        real(default), intent(in) :: sqrts
        real(default), dimension(:, :), intent(inout) :: x
        real(default), dimension(:), intent(inout) :: phs_factor
        real(default), intent(out) :: volume
        integer :: g, t
        integer(TC) :: k, k_root, k_in
        g = forest%grove_lookup (channel)
        t = channel - forest%grove(g)%tree_count_offset
        call phs_prt_set_undefined (forest%prt)
        k_in = forest%n_tot
        forall (k = 1:forest%n_in)
            forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
        end forall
        forall (k = 1:forest%n_out)
            forest%prt(ibset(0,k-1)) = forest%prt_out(k)
        end forall
        call phs_forest_combine_particles (forest)
        call phs_tree_compute_volume &
            (forest%grove(g)%tree(t), sqrts, volume)
        call phs_tree_compute_x_from_momenta &
            (forest%grove(g)%tree(t), &
             forest%prt, phs_factor(channel), sqrts, x(:,channel))
    end subroutine phs_forest_recover_channel

```

### 11.6.13 Test of forest setup

Write a possible phase-space file for a  $2 \rightarrow 3$  process and make the corresponding forest. Print the forest and the resulting VAMP equivalence array. Choose some in-particle momenta and a random-number array and evaluate out-particles and phase-space factors.

```

<XXX PHS forests: public>+≡
    public :: phs_forest_test

<XXX PHS forests: procedures>+≡
    subroutine phs_forest_test ()
        use os_interface, only: os_data_t
        type(os_data_t) :: os_data
        type(phs_forest_t) :: forest
        type(model_t), pointer :: model
        type(string_t) :: process_id
        type(flavor_t), dimension(5) :: flv
        type(vamp_equivalences_t) :: vamp_eq
        type(string_t) :: filename
        type(interaction_t) :: int
        integer :: unit
        integer, parameter :: u = 20
        integer, parameter :: n_dim_extra = 2
        logical, dimension(2), parameter :: &
            externally_generated = (/ .true., .false. /)
        logical, parameter :: azimuthal_dependence = .false.
        type(mapping_defaults_t) :: mapping_defaults
        logical :: found_process, ok
        integer :: channel, ch
        logical, dimension(4) :: active = .true.
        real(default) :: sqrts = 1000
        real(default), dimension(5,4) :: x
        real(default), dimension(4) :: factor
        real(default) :: volume
        print *, "*** Read model file"
        call syntax_model_file_init ()
        call model_list_read_model &
            (var_str("SM"), var_str("SM.mdl"), os_data, model)
        call syntax_model_file_final ()
        print *
        print *, "*** Create phase-space file 'test.phs'"
        call flavor_init (flv, (/ 11, -11, 11, -11, 22 /), model)
        open (file="test.phs", unit=u, action="write")
        write (u, *) "process foo"
        write (u, *) 'md5sum_process      = "6ABA33BC2927925D0F073B1C1170780A"'
        write (u, *) 'md5sum_model_par   = "1A0B151EE6E2DEB92D880320355A3EAB"'
        write (u, *) 'md5sum_phs_config = "B6A8877058809A8BDD54753CDAB83ACE"'
        write (u, *) "sqrts              = 100.00000000000000"
        write (u, *) "m_threshold_s     = 50.00000000000000"
        write (u, *) "m_threshold_t     = 100.00000000000000"
        write (u, *) "off_shell         = 2"
        write (u, *) "t_channel         = 6"
        write (u, *) "keep_nonresonant = F"
        write (u, *) ""
        write (u, *) " grove"

```

```

write (u, *) "      tree 3 7"
write (u, *) "      map 3 s_channel 23"
write (u, *) "      tree 5 7"
write (u, *) "      tree 6 7"
write (u, *) "    grove"
write (u, *) "      tree 9 11"
write (u, *) "      map 9 t_channel 22"
close (u)
print *
print *, "*** Read phase-space file 'test.phs'"
call syntax_phs_forest_init ()
process_id = "foo"
unit = free_unit ()
filename = "test.phs"
call phs_forest_read &
      (forest, filename, process_id, 2, 3, model, found_process)
print *
print *, "*** Set parameters, flavors, equiv, momenta"
call phs_forest_set_flavors (forest, flv)
call phs_forest_set_parameters (forest, mapping_defaults, .false.)
call phs_forest_setup_prt_combinations (forest)
call phs_forest_set_equivalences (forest)
call interaction_init (int, 2, 0, 3)
call interaction_set_momentum (int, &
      vector4_moving (500._default, 500._default, 3), 1)
call interaction_set_momentum (int, &
      vector4_moving (500._default, -500._default, 3), 2)
call phs_forest_set_prt_in (forest, int)
channel = 2
x = 0
x(:,channel) = (/ 0.3, 0.4, 0.1, 0.9, 0.6 /)
1 format (5(1x,G12.5))
print *, "Input values:"
print 1, x(:,channel)
print *
print *, "*** Evaluate phase space"
call phs_forest_evaluate_phase_space (forest, &
      channel, active, sqrts, x, factor, volume, ok)
call phs_forest_get_prt_out (forest, int)
print *, "Output values:"
do ch = 1, 4
      print 1, x(:,ch)
end do
call interaction_write (int)
print *, "factors:"
print 1, factor
print *, "volume:"
print 1, volume
call phs_forest_write (forest)
print *
print *, "*** Compute equivalences"
call phs_forest_setup_vamp_equivalences (forest, &
      n_dim_extra, externally_generated, azimuthal_dependence, &
      vamp_eq)

```

```

    call vamp_equivalences_write (vamp_eq)
    print *
    print *, "*** Cleanup"
    call vamp_equivalences_final (vamp_eq)
    call phs_forest_final (forest)
    call syntax_phs_forest_final ()
end subroutine phs_forest_test

```

## 11.7 Finding phase space parameterizations

If the phase space configuration is not found in the appropriate file, we should generate one.

The idea is to construct all Feynman diagrams subject to certain constraints which eliminate everything that is probably irrelevant for the integration. These Feynman diagrams (cascades) are grouped in groves by finding equivalence classes related by symmetry and ordered with respect to their importance (resonances). Finally, the result (or part of it) is written to file and used for the integration.

This module may eventually disappear and be replaced by CAML code. In particular, we need here a set of Feynman rules (vertices with particle codes, but not the factors). Thus, the module works for the Standard Model only.

Note that this module is stand-alone, it communicates to the main program only via the generated ASCII phase-space configuration file.

```

<cascades.f90>≡
  <File header>

  module cascades

    <Use kinds>
    use kinds, only: TC, i8, i32 !NODEP!
    <Use strings>
    use limits, only: CASCADE_SET_FILL_RATIO, MAX_WARN_RESONANCE !NODEP!
    <Use file utils>
    use diagnostics !NODEP!
    use hashes
    use sorting
    use pdg_arrays, only: UNDEFINED
    use models
    use flavors
    use phs_forests

    <Standard module head>

    <Cascades: public>

    <Cascades: parameters>

    <Cascades: types>

    <Cascades: interfaces>

```

```
contains

  <Cascades: procedures>

end module cascades
```

### 11.7.1 The mapping modes

The valid mapping modes, to be used below. We will make use of the convention that mappings of internal particles have a positive value. Only for positive values, the flavor code is propagated when combining cascades.

```
<Mapping modes>≡
  integer, parameter :: &
    & EXTERNAL_PRT = -1, &
    & NO_MAPPING = 0, S_CHANNEL = 1, T_CHANNEL = 2, U_CHANNEL = 3, &
    & RADIATION = 4, COLLINEAR = 5, INFRARED = 6, &
    & STEP_MAPPING_E = 11, STEP_MAPPING_H = 12

<Cascades: parameters>≡
  <Mapping modes>
```

### 11.7.2 The cascade type

A cascade is essentially the same as a decay tree (both definitions may be merged in a later version). It contains a linked tree of nodes, each of which representing an internal particle. In contrast to decay trees, each node has a definite particle code. These nodes need not be modified, therefore we can use pointers and do not have to copy them. Thus, physically each cascades has only a single node, the mother particle. However, to be able to compare trees quickly, we store in addition an array of binary codes which is always sorted in ascending order. This is accompanied by a corresponding list of particle codes. The index is the location of the corresponding cascade in the cascade set, this may be used to access the daughters directly.

The real mass is the particle mass belonging to the particle code. The minimal mass is the sum of the real masses of all its daughters; this is the kinematical cutoff. The effective mass may be zero if the particle mass is below a certain threshold; it may be the real mass if the particle is resonant; or it may be some other value.

The logical `t_channel` is set if this a  $t$ -channel line, while `initial` is true only for an initial particle. Note that both initial particles are also `t_channel` by definition, and that they are distinguished by the direction of the tree: One of them decays and is the root of the tree, while the other one is one of the leaves.

The cascade is a list of nodes (particles) which are linked via the `daughter` entries. The node is the mother particle of the decay cascade. Much of the information in the nodes is repeated in arrays, to be accessible more easily. The arrays will be kept sorted by binary codes.

The counter `n_off_shell` is increased for each internal line that is neither resonant nor log-enhanced. It is set to zero if the current line is resonant, since this implies on-shell particle production and subsequent decay.

The counter `n_t_channel` is non-negative once an initial particle is included in the tree: then, it counts the number of  $t$ -channel lines.

The `multiplicity` is the number of branchings to follow until all daughters are on-shell. A resonant or non-decaying particle has multiplicity one. Merging nodes, the multiplicities add unless the mother is a resonance. An initial or final node has multiplicity zero.

The arrays correspond to the subnode tree `tree` of the current cascade. PDG codes are stored only for those positions which are resonant, with the exception of the last entry, i.e., the current node. Other positions, in particular external legs, are assigned undefined PDG code.

A cascade is uniquely identified by its tree, the tree of PDG codes, and the tree of mappings. The tree of resonances is kept only to mask the PDG tree as described above.

$\langle \text{Cascades: types} \rangle \equiv$

```

type :: cascade_t
  private
  ! counters
  integer :: index = 0
  integer :: grove = 0
  ! status
  logical :: active = .false.
  logical :: complete = .false.
  logical :: incoming = .false.
  ! this node
  integer(TC) :: bincode = 0
  type(flavor_t) :: flv
  integer :: pdg = UNDEFINED
  logical :: is_vector = .false.
  real(default) :: m_min = 0
  real(default) :: m_rea = 0
  real(default) :: m_eff = 0
  integer :: mapping = NO_MAPPING
  logical :: on_shell = .false.
  logical :: resonant = .false.
  logical :: log_enhanced = .false.
  logical :: t_channel = .false.
  ! global tree properties
  integer :: multiplicity = 0
  integer :: internal = 0
  integer :: n_off_shell = 0
  integer :: n_resonances = 0
  integer :: n_log_enhanced = 0
  integer :: n_t_channel = 0
  integer :: res_hash = 0
  ! the sub-node tree
  integer :: depth = 0
  integer(TC), dimension(:), allocatable :: tree
  integer, dimension(:), allocatable :: tree_pdg
  integer, dimension(:), allocatable :: tree_mapping
  logical, dimension(:), allocatable :: tree_resonant
  ! branch connections
  logical :: has_children = .false.
  type(cascade_t), pointer :: daughter1 => null ()

```

```

        type(cascade_t), pointer :: daughter2 => null ()
        type(cascade_t), pointer :: mother => null ()
        ! next in list
        type(cascade_t), pointer :: next => null ()
    end type cascade_t

```

```

<Cascades: procedures>≡
    subroutine cascade_init (cascade, depth)
        type(cascade_t), intent(out) :: cascade
        integer, intent(in) :: depth
        integer, save :: index = 0
        index = cascade_index ()
        cascade%index = index
        cascade%depth = depth
        cascade%active = .true.
        allocate (cascade%tree (depth))
        allocate (cascade%tree_pdg (depth))
        allocate (cascade%tree_mapping (depth))
        allocate (cascade%tree_resonant (depth))
    end subroutine cascade_init

```

Keep and increment a global index

```

<Cascades: procedures>+≡
    function cascade_index (seed) result (index)
        integer :: index
        integer, intent(in), optional :: seed
        integer, save :: i = 0
        if (present (seed)) i = seed
        i = i + 1
        index = i
    end function cascade_index

```

We need three versions of writing cascades. This goes to the phase-space file:

```

<Cascades: procedures>+≡
    subroutine cascade_write_file_format (cascade, model, unit)
        type(cascade_t), intent(in) :: cascade
        type(model_t), intent(in), target :: model
        integer, intent(in), optional :: unit
        type(flavor_t) :: flv
        integer :: u, i
1   format(3x,A,1x,40(1x,I4))
2   format(3x,A,1x,I3,1x,A,1x,I7,1x,'!',1x,A)
        u = output_unit (unit); if (u < 0) return
    !   write (u, 1) "tree", reduced (cascade%tree)
        call write_reduced (cascade%tree, u)
        write (u, "(A)")
        do i = 1, cascade%depth
            call flavor_init (flv, cascade%tree_pdg(i), model)
            select case (cascade%tree_mapping(i))
            case (NO_MAPPING, EXTERNAL_PRT)
            case (S_CHANNEL)
                write(u,2) 'map', &
                    cascade%tree(i), 's_channel', abs (cascade%tree_pdg(i)), &

```

```

        char (flavor_get_name (flv))
case (T_CHANNEL)
    write(u,2) 'map', &
        cascade%tree(i), 't_channel', abs (cascade%tree_pdg(i)), &
        char (flavor_get_name (flv))
case (U_CHANNEL)
    write(u,2) 'map', &
        cascade%tree(i), 'u_channel', abs (cascade%tree_pdg(i)), &
        char (flavor_get_name (flv))
case (RADIATION)
    write(u,2) 'map', &
        cascade%tree(i), 'radiation', abs (cascade%tree_pdg(i)), &
        char (flavor_get_name (flv))
case (COLLINEAR)
    write(u,2) 'map', &
        cascade%tree(i), 'collinear', abs (cascade%tree_pdg(i)), &
        char (flavor_get_name (flv))
case (INFRARED)
    write(u,2) 'map', &
        cascade%tree(i), 'infrared ', abs (cascade%tree_pdg(i)), &
        char (flavor_get_name (flv))
case default
    call msg_bug (" Impossible mapping mode encountered")
end select
end do
contains
subroutine write_reduced (array, unit)
    integer(TC), dimension(:), intent(in) :: array
    integer, intent(in) :: unit
    integer :: i
    write (u, "(3x,A,1x)", advance="no") "tree"
    do i = 1, size (array)
        if (decay_level (array(i)) > 1) then
            write (u, "(1x,I0)", advance="no") array(i)
        end if
    end do
end subroutine write_reduced
! ICE in gfortran 4.6.0
! function reduced (array)
!   integer(TC), dimension(:), allocatable :: reduced
!   integer(TC), dimension(:), intent(in) :: array
!   logical, dimension(size(array)) :: mask
!   mask = decay_level (array) > 1
!   allocate (reduced (count (mask)))
!   reduced = pack (array, mask)
! end function reduced
elemental function decay_level (k) result (l)
    integer(TC), intent(in) :: k
    integer :: l
    integer :: i
    l = 0
    do i = 0, bit_size(k) - 1
        if (btest(k,i)) l = l + 1
    end do

```



```

end function decay_level
subroutine start_comment (u)
  integer, intent(in) :: u
  write(u, '(1x,A)', advance='no') '!'
end subroutine start_comment
end subroutine cascade_write_file_format

```

This creates metapost source for graphical display:

*(Cascades: procedures)+≡*

```

subroutine cascade_write_graph_format (cascade, count, unit)
  type(cascade_t), intent(in) :: cascade
  integer, intent(in) :: count
  integer, intent(in), optional :: unit
  integer :: u
  integer(TC) :: mask
  type(string_t) :: left_str, right_str
  u = output_unit (unit); if (u < 0) return
  mask = 2**((cascade%depth+3)/2) - 1
  left_str = ""
  right_str = ""
  write (u, '(A)') "\begin{minipage}{105pt}"
  write (u, '(A)') "\vspace{30pt}"
  write (u, '(A)') "\begin{center}"
  write (u, '(A)') "\begin{fmfgraph*}(55,55)"
  call graph_write (cascade, mask)
  write (u, '(A)') "\fmfleft{" // char (extract (left_str, 2)) // "}"
  write (u, '(A)') "\fmfright{" // char (extract (right_str, 2)) // "}"
  write (u, '(A)') "\end{fmfgraph*}\\"
  write (u, '(A,I5,A)') "\fbox{$", count, "$}"
  write (u, '(A)') "\end{center}"
  write (u, '(A)') "\end{minipage}"
  write (u, '(A)') "%"

```

contains

```

recursive subroutine graph_write (cascade, mask, reverse)
  type(cascade_t), intent(in) :: cascade
  integer(TC), intent(in) :: mask
  logical, intent(in), optional :: reverse
  logical :: rev
  rev = .false.; if (present(reverse)) rev = reverse
  if (cascade%has_children) then
    if (.not.rev) then
      call vertex_write (cascade, cascade%daughter1, mask)
      call vertex_write (cascade, cascade%daughter2, mask)
    else
      call vertex_write (cascade, cascade%daughter2, mask, .true.)
      call vertex_write (cascade, cascade%daughter1, mask, .true.)
    end if
    if (cascade%complete) then
      call vertex_write (cascade, cascade%mother, mask, .true.)
      write (u, '(A,I0,A)') "\fmfv{d.shape=square}{v0}"
    end if
  else
    if (cascade%incoming) then
      call external_write (cascade%bincode, &

```

```

        flavor_get_tex_name (flavor_anti (cascade%flv)), left_str)
    else
        call external_write (cascade%bincode, &
            flavor_get_tex_name (cascade%flv), right_str)
    end if
end if
end subroutine graph_write
recursive subroutine vertex_write (cascade, daughter, mask, reverse)
    type(cascade_t), intent(in) :: cascade, daughter
    integer(TC), intent(in) :: mask
    logical, intent(in), optional :: reverse
    integer :: bincode
    if (cascade%complete) then
        bincode = 0
    else
        bincode = cascade%bincode
    end if
    call graph_write (daughter, mask, reverse)
    if (daughter%has_children) then
        call line_write (bincode, daughter%bincode, daughter%flv, &
            mapping=daughter%mapping)
    else
        call line_write (bincode, daughter%bincode, daughter%flv)
    end if
end subroutine vertex_write
subroutine line_write (i1, i2, flv, mapping)
    integer(TC), intent(in) :: i1, i2
    type(flavor_t), intent(in) :: flv
    integer, intent(in), optional :: mapping
    integer :: k1, k2
    type(string_t) :: prt_type
    select case (flavor_get_spin_type (flv))
    case (SCALAR);      prt_type = "plain"
    case (SPINOR);      prt_type = "fermion"
    case (VECTOR);      prt_type = "boson"
    case (VECTORSPINOR); prt_type = "fermion"
    case (TENSOR);      prt_type = "dbl_wiggly"
    case default;       prt_type = "dashes"
    end select
    if (flavor_is_antiparticle (flv)) then
        k1 = i2; k2 = i1
    else
        k1 = i1; k2 = i2
    end if
    if (present (mapping)) then
        select case (mapping)
        case (S_CHANNEL)
            write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
                & ",f=blue,lab=\sm\blue$" // &
                & char (flavor_get_tex_name (flv)) // "$}" // &
                & "{v", k1, ",v", k2, "}"
        case (T_CHANNEL, U_CHANNEL)
            write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
                & ",f=cyan,lab=\sm\cyan$" // &

```

```

        & char (flavor_get_tex_name (flv)) // "$" // &
        & "{v", k1, ",v", k2, "}"
    case (RADIATION)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
        & ",f=green,lab=\sm\green$" // &
        & char (flavor_get_tex_name (flv)) // "$" // &
        & "{v", k1, ",v", k2, "}"
    case (COLLINEAR)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
        & ",f=magenta,lab=\sm\magenta$" // &
        & char (flavor_get_tex_name (flv)) // "$" // &
        & "{v", k1, ",v", k2, "}"
    case (INFRARED)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
        & ",f=red,lab=\sm\red$" // &
        & char (flavor_get_tex_name (flv)) // "$" // &
        & "{v", k1, ",v", k2, "}"
    case default
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
        & ",f=black}" // &
        & "{v", k1, ",v", k2, "}"
    end select
else
    write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
    & "}" // &
    & "{v", k1, ",v", k2, "}"
end if
end subroutine line_write
subroutine external_write (bincode, name, ext_str)
    integer(TC), intent(in) :: bincode
    type(string_t), intent(in) :: name
    type(string_t), intent(inout) :: ext_str
    character(len=20) :: str
    write (str, '(A2,I0)') ",v", bincode
    ext_str = ext_str // trim (str)
    write (u, '(A,I0,A,I0,A)') "\fmflabel{\sm$" &
    // char (name) &
    // "\",(" , bincode, ")" &
    // "$}{v", bincode, "}"
end subroutine external_write
end subroutine cascade_write_graph_format

```

This is for screen/debugging output:

*(Cascades: procedures)+≡*

```

subroutine cascade_write (cascade, unit)
    type(cascade_t), intent(in) :: cascade
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, *) 'Cascade #', cascade%index
    write (u, *) ' Grove:      #', cascade%grove
    write (u, *) ' act/cmp/inc: ', &
        cascade%active, cascade%complete, cascade%incoming
    write (u, *) ' Bincode:      ', cascade%bincode

```

```

write (u, "(1x,A)", advance="no") ' Flavor:      '
call flavor_write (cascade%flv, unit)
write (u, *) ' Active flavor:', cascade%pdg
write (u, *) ' Is vector:      ', cascade%is_vector
write (u, *) ' Mass (m/r/e): ', &
    cascade%m_min, cascade%m_rea, cascade%m_eff
write (u, *) ' Mapping:      ', cascade%mapping
write (u, *) ' res/log/tch: ', &
    cascade%resonant, cascade%log_enhanced, cascade%t_channel
write (u, *) ' Multiplicity: ', cascade%multiplicity
write (u, *) ' n intern/off: ', cascade%internal, cascade%n_off_shell
write (u, *) ' n res/log/tch:', &
    cascade%n_resonances, cascade%n_log_enhanced, cascade%n_t_channel
write (u, *) ' Depth:      ', cascade%depth
write (u, *) ' Tree:      ', cascade%tree
write (u, *) ' Tree(PDG):  ', cascade%tree_pdg
write (u, *) ' Tree(mapping):', cascade%tree_mapping
write (u, *) ' Tree(res):  ', cascade%tree_resonant
if (cascade%has_children) then
    write (u, *) ' Daughter1/2: ', &
        cascade%daughter1%index, cascade%daughter2%index
end if
if (associated (cascade%mother)) then
    write (u, *) ' Mother:      ', cascade%mother%index
end if
end subroutine cascade_write

```

### 11.7.3 Creating new cascades

This initializes a single-particle cascade (external, final state). The PDG entry in the tree is set undefined because the cascade is not resonant. However, the flavor entry is set, so the cascade flavor is identified nevertheless.

*(Cascades: procedures)+≡*

```

subroutine cascade_init_outgoing (cascade, flv, pos, m_thr)
    type(cascade_t), intent(out) :: cascade
    type(flavor_t), intent(in) :: flv
    integer, intent(in) :: pos
    real(default), intent(in) :: m_thr
    call cascade_init (cascade, 1)
    cascade%bincode = ibset (0_TC, pos-1)
    cascade%flv = flv
    cascade%pdg = abs (flavor_get_pdg (cascade%flv))
    cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
    cascade%m_min = flavor_get_mass (flv)
    cascade%m_rea = cascade%m_min
    if (cascade%m_rea >= m_thr) then
        cascade%m_eff = cascade%m_rea
    end if
    cascade%on_shell = .true.
    cascade%multiplicity = 1
    cascade%tree(1) = cascade%bincode
    cascade%tree_pdg(1) = cascade%pdg

```

```

        cascade%tree_mapping(1) = EXTERNAL_PRT
        cascade%tree_resonant(1) = .false.
    end subroutine cascade_init_outgoing

```

The same for an incoming line:

```

<Cascades: procedures>+=
    subroutine cascade_init_incoming (cascade, flv, pos, m_thr)
        type(cascade_t), intent(out) :: cascade
        type(flavor_t), intent(in) :: flv
        integer, intent(in) :: pos
        real(default), intent(in) :: m_thr
        call cascade_init (cascade, 1)
        cascade%incoming = .true.
        cascade%bincode = ibset (0_TC, pos-1)
        cascade%flv = flavor_anti (flv)
        cascade%pdg = abs (flavor_get_pdg (flv))
        cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
        cascade%m_min = flavor_get_mass (flv)
        cascade%m_rea = cascade%m_min
        if (cascade%m_rea >= m_thr) then
            cascade%m_eff = cascade%m_rea
        end if
        cascade%on_shell = .true.
        cascade%n_t_channel = 0
        cascade%n_off_shell = 0
        cascade%tree(1) = cascade%bincode
        cascade%tree_pdg(1) = cascade%pdg
        cascade%tree_mapping(1) = EXTERNAL_PRT
        cascade%tree_resonant(1) = .false.
    end subroutine cascade_init_incoming

```

#### 11.7.4 Tools

This function returns true if the two cascades share no common external particle. This is a requirement for joining them.

```

<Cascades: interfaces>=
    interface operator(.disjunct.)
        module procedure cascade_disjunct
    end interface

<Cascades: procedures>+=
    function cascade_disjunct (cascade1, cascade2) result (flag)
        logical :: flag
        type(cascade_t), intent(in) :: cascade1, cascade2
        flag = iand (cascade1%bincode, cascade2%bincode) == 0
    end function cascade_disjunct

```

Compute a hash code for the resonance pattern of a cascade. We count the number of times each particle appears as a resonance.

We pack the PDG codes of the resonances in two arrays (s-channel and t-channel), sort them both, concatenate the results, transfer to i8 integers, and compute the hash code from this byte stream.

```

<Cascades: procedures>+=
  subroutine cascade_assign_resonance_hash (cascade)
    type(cascade_t), intent(inout) :: cascade
    integer(i8), dimension(1) :: mold
    cascade%res_hash = hash (transfer &
      (concat (sort (pack (cascade%tree_pdg, &
        cascade%tree_resonant)), &
        sort (pack (cascade%tree_pdg, &
          cascade%tree_mapping == T_CHANNEL .or. &
          cascade%tree_mapping == U_CHANNEL))), &
        mold))
  end subroutine cascade_assign_resonance_hash

```

### 11.7.5 Hash entries for cascades

We will set up a hash array which contains keys of and pointers to cascades. We hold a list of cascade (pointers) within each bucket. This is not for collision resolution, but for keeping similar, but unequal cascades together.

```

<Cascades: types>+=
  type :: cascade_p
    type(cascade_t), pointer :: cascade => null ()
    type(cascade_p), pointer :: next => null ()
  end type cascade_p

```

Here is the bucket or hash entry type:

```

<Cascades: types>+=
  type :: hash_entry_t
    integer(i32) :: hashval = 0
    integer(i8), dimension(:), allocatable :: key
    type(cascade_p), pointer :: first => null ()
    type(cascade_p), pointer :: last => null ()
  end type hash_entry_t

```

Finalize: just deallocate the list; the contents are just pointers.

```

<Cascades: procedures>+=
  subroutine hash_entry_final (hash_entry)
    type(hash_entry_t), intent(inout) :: hash_entry
    type(cascade_p), pointer :: current
    do while (associated (hash_entry%first))
      current => hash_entry%first
      hash_entry%first => current%next
      deallocate (current)
    end do
  end subroutine hash_entry_final

```

Output: concise format for debugging, just list cascade indices.

```

(Cascades: procedures)+=≡
subroutine hash_entry_write (hash_entry, unit)
  type(hash_entry_t), intent(in) :: hash_entry
  integer, intent(in), optional :: unit
  type(cascade_p), pointer :: current
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, "(1x,A)", advance="no") "Entry:"
  do i = 1, size (hash_entry%key)
    write (u, "(1x,I0)", advance="no") hash_entry%key(i)
  end do
  write (u, "(1x,A)", advance="no") "->"
  current => hash_entry%first
  do while (associated (current))
    write (u, "(1x,I7)", advance="no") current%cascade%index
    current => current%next
  end do
  write (u, *)
end subroutine hash_entry_write

```

This function adds a cascade pointer to the bucket. If ok is present, check first if it is already there and return failure if yes. If `cascade_ptr` is also present, set it to the current cascade if successful. If not, set it to the cascade that is already there.

```

(Cascades: procedures)+=≡
subroutine hash_entry_add_cascade_ptr (hash_entry, cascade, ok, cascade_ptr)
  type(hash_entry_t), intent(inout) :: hash_entry
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out), optional :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_p), pointer :: current
  if (present (ok)) then
    call hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
    if (.not. ok) return
  end if
  allocate (current)
  current%cascade => cascade
  if (associated (hash_entry%last)) then
    hash_entry%last%next => current
  else
    hash_entry%first => current
  end if
  hash_entry%last => current
end subroutine hash_entry_add_cascade_ptr

```

This function checks whether a cascade is already in the bucket. For incomplete cascades, we look for an exact match. It should suffice to verify the tree, the PDG codes, and the mapping modes. This is the information that is written to the phase space file.

For complete cascades, we ignore the PDG code at positions with mappings infrared, collinear, or t/u-channel. Thus a cascade which is distinguished only by

PDG code at such places, is flagged existent. If the convention is followed that light particles come before heavier ones (in the model definition), this ensures that the lightest particle is kept in the appropriate place, corresponding to the strongest peak.

For external cascades (incoming/outgoing) we take the PDG code into account even though it is zeroed in the PDG-code tree.

```

<Cascades: procedures>+=
subroutine hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
  type(hash_entry_t), intent(in), target :: hash_entry
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_p), pointer :: current
  integer, dimension(:), allocatable :: tree_pdg
  ok = .true.
  allocate (tree_pdg (size (cascade%tree_pdg)))
  if (cascade%complete) then
    where (cascade%tree_mapping == INFRARED .or. &
           cascade%tree_mapping == COLLINEAR .or. &
           cascade%tree_mapping == T_CHANNEL .or. &
           cascade%tree_mapping == U_CHANNEL)
      tree_pdg = 0
    elsewhere
      tree_pdg = cascade%tree_pdg
    end where
  else
    tree_pdg = cascade%tree_pdg
  end if
  current => hash_entry%first
  do while (associated (current))
    if (current%cascade%depth == cascade%depth) then
      if (all (current%cascade%tree == cascade%tree)) then
        if (all (current%cascade%tree_mapping == cascade%tree_mapping)) &
          then
            if (all (current%cascade%tree_pdg .match. tree_pdg)) then
              if (present (cascade_ptr)) cascade_ptr => current%cascade
              ok = .false.; return
            end if
          end if
        end if
      end if
    end if
    current => current%next
  end do
  if (present (cascade_ptr)) cascade_ptr => cascade
end subroutine hash_entry_check_cascade

```

For PDG codes, we specify that the undefined code matches any code. This is already defined for flavor objects, but here we need it for the codes themselves.

```

<Cascades: interfaces>+=
interface operator(.match.)
  module procedure pdg_match
end interface

```



```

<Cascades: procedures>+=
  elemental function pdg_match (pdg1, pdg2) result (flag)
    logical :: flag
    integer(TC), intent(in) :: pdg1, pdg2
    select case (pdg1)
    case (0)
      flag = .true.
    case default
      select case (pdg2)
      case (0)
        flag = .true.
      case default
        flag = pdg1 == pdg2
      end select
    end select
  end function pdg_match

```

### 11.7.6 The cascade set

The cascade set will later be transformed into the decay forest. It is set up as a linked list. In addition to the usual **first** and **last** pointers, there is a **first\_t** pointer which points to the first t-channel cascade (after all s-channel cascades), and a **first\_k** pointer which points to the first final cascade (with a keystone).

As an auxiliary device, the object contains a hash array with associated parameters where an additional pointer is stored for each cascade. The keys are made from the relevant cascade data. This hash is used for fast detection (and thus avoidance) of double entries in the cascade list.

```

<Cascades: public>≡
  public :: cascade_set_t

<Cascades: types>+=
  type :: cascade_set_t
    private
    type(model_t), pointer :: model
    integer :: n_in, n_out, n_tot
    type(flavor_t), dimension(:, :), allocatable :: flv
    integer :: depth_out, depth_tot
    real(default) :: sqrts = 0
    real(default) :: m_threshold_s = 0
    real(default) :: m_threshold_t = 0
    integer :: off_shell = 0
    integer :: t_channel = 0
    logical :: keep_nonresonant
    integer :: n_groves = 0
    ! The cascade list
    type(cascade_t), pointer :: first => null ()
    type(cascade_t), pointer :: last => null ()
    type(cascade_t), pointer :: first_t => null ()
    type(cascade_t), pointer :: first_k => null ()
    ! The hashtable
    integer :: n_entries = 0
    real :: fill_ratio = 0

```

```

integer :: n_entries_max = 0
integer(i32) :: mask = 0
logical :: fatal_beam_decay = .true.
type(hash_entry_t), dimension(:), allocatable :: entry
end type cascade_set_t

```

Return true if there are cascades which are active and complete, so the phase space file would be nonempty.

```

<Cascades: public>+≡
public :: cascade_set_is_valid

<Cascades: procedures>+≡
function cascade_set_is_valid (cascade_set) result (flag)
logical :: flag
type(cascade_set_t), intent(in) :: cascade_set
type(cascade_t), pointer :: cascade
flag = .false.
cascade => cascade_set%first_k
do while (associated (cascade))
    if (cascade%active .and. cascade%complete) then
        flag = .true.
        return
    end if
    cascade => cascade%next
end do
end function cascade_set_is_valid

```

The initializer sets up the hash table with some initial size guessed by looking at the number of external particles. We choose 256 for 3 external particles and a factor of 4 for each additional particle, limited at  $2^{30}=1\text{G}$ .

```

<Limits: public parameters>+≡
real, parameter, public :: CASCADE_SET_FILL_RATIO = 0.1

<Cascades: procedures>+≡
subroutine cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
    fatal_beam_decay, flv)
type(cascade_set_t), intent(out) :: cascade_set
type(model_t), intent(in), target :: model
integer, intent(in) :: n_in, n_out
type(phs_parameters_t), intent(in) :: phs_par
logical, intent(in) :: fatal_beam_decay
type(flavor_t), dimension(:,:), intent(in), optional :: flv
integer :: size_guess
cascade_set%model => model
cascade_set%n_in = n_in
cascade_set%n_out = n_out
cascade_set%n_tot = n_in + n_out
if (present (flv)) then
    allocate (cascade_set%flv (size (flv, 1), size (flv, 2)))
    call flavor_init (cascade_set%flv, flavor_get_pdg (flv), model)
end if
select case (n_in)
case (1); cascade_set%depth_out = 2 * n_out - 3
case (2); cascade_set%depth_out = 2 * n_out - 1

```

```

end select
cascade_set%depth_tot = 2 * cascade_set%n_tot - 3
cascade_set%sqrts = phs_par%sqrts
cascade_set%m_threshold_s = phs_par%m_threshold_s
cascade_set%m_threshold_t = phs_par%m_threshold_t
cascade_set%off_shell = phs_par%off_shell
cascade_set%t_channel = phs_par%t_channel
cascade_set%keep_nonresonant = phs_par%keep_nonresonant
cascade_set%fill_ratio = CASCADE_SET_FILL_RATIO
size_guess = ishft (256, min (2 * (cascade_set%n_tot - 3), 22))
cascade_set%n_entries_max = size_guess * cascade_set%fill_ratio
cascade_set%mask = size_guess - 1
allocate (cascade_set%entry (0:cascade_set%mask))
cascade_set%fatal_beam_decay = fatal_beam_decay
end subroutine cascade_set_init

```

The finalizer has to delete both the hash and the list. We assume that the hash only contains pointers, so it is simply deallocated, while the list entries are physically deleted.

```

<Cascades: public>+≡
public :: cascade_set_final

<Cascades: procedures>+≡
subroutine cascade_set_final (cascade_set)
type(cascade_set_t), intent(inout), target :: cascade_set
type(cascade_t), pointer :: current
if (allocated (cascade_set%entry)) deallocate (cascade_set%entry)
do while (associated (cascade_set%first))
current => cascade_set%first
cascade_set%first => cascade_set%first%next
deallocate (current)
end do
end subroutine cascade_set_final

```

Write the process in ASCII format, in columns that are headed by the corresponding bincode.

```

<Cascades: public>+≡
public :: cascade_set_write_process_bincode_format

<Cascades: procedures>+≡
subroutine cascade_set_write_process_bincode_format (cascade_set, unit)
type(cascade_set_t), intent(in), target :: cascade_set
integer, intent(in), optional :: unit
integer, dimension(:), allocatable :: bincode, field_width
integer :: n_in, n_out, n_tot, n_flg
integer :: u, f, i, bc
character(20) :: str
type(string_t) :: fmt_head
type(string_t), dimension(:), allocatable :: fmt_proc
u = output_unit (unit); if (u < 0) return
if (.not. allocated (cascade_set%flv)) return
write (u, "('!',1x,A)") "List of subprocesses with particle bincodes:"
n_in = cascade_set%n_in
n_out = cascade_set%n_out

```

```

n_tot = cascade_set%n_tot
n_flv = size (cascade_set%flv, 2)
allocate (bincode (n_tot), field_width (n_tot), fmt_proc (n_tot))
bc = 1
do i = 1, n_out
    bincode(n_in + i) = bc
    bc = 2 * bc
end do
do i = n_in, 1, -1
    bincode(i) = bc
    bc = 2 * bc
end do
do i = 1, n_tot
    write (str, "(I0)") bincode(i)
    field_width(i) = len_trim (str)
    do f = 1, n_flv
        field_width(i) = max (field_width(i), &
            len (flavor_get_name (cascade_set%flv(i,f))))
    end do
end do
fmt_head = "('!'"
do i = 1, n_tot
    fmt_head = fmt_head // ",1x,"
    fmt_proc(i) = "(1x,"
    write (str, "(I0)") field_width(i)
    fmt_head = fmt_head // "I" // trim(str)
    fmt_proc(i) = fmt_proc(i) // "A" // trim(str)
    if (i == n_in) then
        fmt_head = fmt_head // ",1x,' '"
    end if
end do
fmt_proc = fmt_proc // ")"
fmt_head = fmt_head // ")"
write (u, char (fmt_head)) bincode
do f = 1, n_flv
    write (u, "('!'", advance="no")
    do i = 1, n_tot
        write (u, char (fmt_proc(i)), advance="no") &
            char (flavor_get_name (cascade_set%flv(i,f)))
        if (i == n_in) write (u, "(1x,'=>'", advance="no")
    end do
    write (u, *)
end do
write (u, char (fmt_head)) bincode
end subroutine cascade_set_write_process_bincode_format

```

Write the process as a L<sup>A</sup>T<sub>E</sub>X expression.

(*Cascades: procedures*) $\vdash$

```

subroutine cascade_set_write_process_tex_format (cascade_set, unit)
    type(cascade_set_t), intent(in), target :: cascade_set
    integer, intent(in), optional :: unit
    integer :: u, f, i
    u = output_unit (unit); if (u < 0) return

```

```

if (.not. allocated (cascade_set%flv)) return
write (u, "(A)") "\begin{align*}"
do f = 1, size (cascade_set%flv, 2)
  do i = 1, cascade_set%n_in
    if (i > 1) write (u, "(A)", advance="no") "\quad "
    write (u, "(A)", advance="no") &
      char (flavor_get_tex_name (cascade_set%flv(i,f)))
  end do
  write (u, "(A)", advance="no") "\quad &\to\quad "
  do i = cascade_set%n_in + 1, cascade_set%n_tot
    if (i > cascade_set%n_in + 1) write (u, "(A)", advance="no") "\quad "
    write (u, "(A)", advance="no") &
      char (flavor_get_tex_name (cascade_set%flv(i,f)))
  end do
  if (f < size (cascade_set%flv, 2)) then
    write (u, "(A)") "\\ "
  else
    write (u, "(A)") ""
  end if
end do
write (u, "(A)") "\end{align*}"
end subroutine cascade_set_write_process_tex_format

```

Three output routines: phase-space file, graph source code, and screen output.

This version generates the phase space file. It deals only with complete cascades.

*<Cascades: public>+≡*

```
public :: cascade_set_write_file_format
```

*<Cascades: procedures>+≡*

```

subroutine cascade_set_write_file_format (cascade_set, unit)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: unit
  type(cascade_t), pointer :: cascade
  integer :: u, grove, count
  logical :: first_in_grove
  u = output_unit (unit); if (u < 0) return
  count = 0
  do grove = 1, cascade_set%n_groves
    first_in_grove = .true.
    cascade => cascade_set%first_k
    do while (associated (cascade))
      if (cascade%active .and. cascade%complete) then
        if (cascade%grove == grove) then
          if (first_in_grove) then
            first_in_grove = .false.
            write (u, *)
            write (u, "(1x,'!',1x,A,1x,I0,A)", advance='no') &
              'Multiplicity =', cascade%multiplicity, ","
            select case (cascade%n_resonances)
            case (0)
              write (u, '(1x,A)', advance='no') 'no resonances, '
            case (1)
              write (u, '(1x,A)', advance='no') '1 resonance, '

```

```

        case default
            write (u, '(1x,I0,1x,A)', advance='no') &
                cascade%n_resonances, 'resonances, '
        end select
        write (u, '(1x,I0,1x,A)', advance='no') &
            cascade%n_log_enhanced, 'logs, '
        write (u, '(1x,I0,1x,A)', advance='no') &
            cascade%n_off_shell, 'off-shell, '
        select case (cascade%n_t_channel)
        case (0); write (u, '(1x,A)') 's-channel graph'
        case (1); write (u, '(1x,A)') '1 t-channel line'
        case default
            write(u, '(1x,I0,1x,A)') &
                cascade%n_t_channel, 't-channel lines'
        end select
        write (u, '(1x,A,I0)') 'grove #', grove
    end if
    count = count + 1
    write (u, "(1x,'!',1x,A,I0)") "Channel #", count
    call cascade_write_file_format (cascade, cascade_set%model, u)
end if
end if
cascade => cascade%next
end do
end do
end subroutine cascade_set_write_file_format

```

This is the graph output format, the driver-file

```

<Cascades: public>+≡
    public :: cascade_set_write_graph_format

<Cascades: procedures>+≡
    subroutine cascade_set_write_graph_format &
        (cascade_set, filename, process_id, unit)
        type(cascade_set_t), intent(in), target :: cascade_set
        type(string_t), intent(in) :: filename, process_id
        integer, intent(in), optional :: unit
        type(cascade_t), pointer :: cascade
        integer :: u, grove, count, pgcount
        logical :: first_in_grove
        u = output_unit (unit); if (u < 0) return
        write (u, '(A)') "\documentclass[10pt]{article}"
        write (u, '(A)') "\usepackage{amsmath}"
        write (u, '(A)') "\usepackage{feynmp}"
        write (u, '(A)') "\usepackage{url}"
        write (u, '(A)') "\usepackage{color}"
        write (u, *)
        write (u, '(A)') "\textwidth 18.5cm"
        write (u, '(A)') "\evensidemargin -1.5cm"
        write (u, '(A)') "\oddsidemargin -1.5cm"
        write (u, *)
        write (u, '(A)') "\newcommand{\blue}{\color{blue}}"
        write (u, '(A)') "\newcommand{\green}{\color{green}}"
        write (u, '(A)') "\newcommand{\red}{\color{red}}"
    end subroutine

```

```

write (u, '(A)') "\newcommand{\magenta}{\color{magenta}}"
write (u, '(A)') "\newcommand{\cyan}{\color{cyan}}"
write (u, '(A)') "\newcommand{\sm}{\footnotesize}"
write (u, '(A)') "\setlength{\parindent}{0pt}"
write (u, '(A)') "\setlength{\parsep}{20pt}"
write (u, *)
write (u, '(A)') "\begin{document}"
write (u, '(A)') "\begin{fmffile}{ " // char (filename) // "}"
write (u, '(A)') "\fmfcmd{color magenta; magenta = red + blue;}"
write (u, '(A)') "\fmfcmd{color cyan; cyan = green + blue;}"
write (u, '(A)') "\begin{fmfshrink}{0.5}"
write (u, '(A)') "\begin{flushleft}"
write (u, *)
write (u, '(A)') "\noindent" // &
& "\textbf{\large\texttt{WHIZARD}} phase space channels" // &
& "\hfill\today"
write (u, *)
write (u, '(A)') "\vspace{10pt}"
write (u, '(A)') "\noindent" // &
& "\textbf{Process:} \url{ " // char (process_id) // "}"
call cascade_set_write_process_tex_format (cascade_set, u)
write (u, *)
write (u, '(A)') "\noindent" // &
& "\textbf{Note:} These are pseudo Feynman graphs that " // &
& "visualize phase-space parameterizations " // &
& "(‘‘integration channels’’). " // &
& "They do \emph{not} indicate Feynman graphs used for the " // &
& "matrix element."
write (u, *)
write (u, '(A)') "\textbf{Color code:} " // &
& "{\blue resonance,} " // &
& "{\cyan t-channel,} " // &
& "{\green radiation,} " // &
& "{\red infrared,} " // &
& "{\magenta collinear,} " // &
& "external/off-shell"
write (u, *)
write (u, '(A)') "\noindent" // &
& "\textbf{Black square:} Keystone, indicates ordering of " // &
& "phase space parameters."
write (u, *)
write (u, '(A)') "\vspace{-20pt}"
count = 0
pgcount = 0
do grove = 1, cascade_set%n_groves
  first_in_grove = .true.
  cascade => cascade_set%first
  do while (associated (cascade))
    if (cascade%active .and. cascade%complete) then
      if (cascade%grove == grove) then
        if (first_in_grove) then
          first_in_grove = .false.
          write (u, *)
          write (u, '(A)') "\vspace{20pt}"

```

```

        write (u, '(A)') "\begin{tabular}{l}"
        write (u, '(A,I5,A)') &
            & "\fbox{\bf Grove \boldmath$, grove, "$} \\[10pt]"
        write (u, '(A,I1,A)') "Multiplicity: ", &
            cascade%multiplicity, "\\"
        write (u, '(A,I1,A)') "Resonances: ", &
            cascade%n_resonances, "\\"
        write (u, '(A,I1,A)') "Log-enhanced: ", &
            cascade%n_log_enhanced, "\\"
        write (u, '(A,I1,A)') "Off-shell: ", &
            cascade%n_off_shell, "\\"
        write (u, '(A,I1,A)') "t-channel: ", &
            cascade%n_t_channel, ""
        write (u, '(A)') "\end{tabular}"
    end if
    count = count + 1
    call cascade_write_graph_format (cascade, count, unit)
    if (pgcount >= 250) then
        write (u, '(A)') "\clearpage"
        pgcount = 0
    end if
end if
end if
end if
cascade => cascade%next
end do
end do
write (u, '(A)') "\end{flushleft}"
write (u, '(A)') "\end{fmfshrink}"
write (u, '(A)') "\end{fmffile}"
write (u, '(A)') "\end{document}"
end subroutine cascade_set_write_graph_format

```

This is for screen output and debugging:

```

<Cascades: public>+≡
    public :: cascade_set_write

<Cascades: procedures>+≡
    subroutine cascade_set_write (cascade_set, unit, active_only, complete_only)
        type(cascade_set_t), intent(in), target :: cascade_set
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: active_only, complete_only
        logical :: active, complete
        type(cascade_t), pointer :: cascade
        integer :: u, i
        u = output_unit (unit); if (u < 0) return
        active = .true.; if (present (active_only)) active = active_only
        complete = .false.; if (present (complete_only)) complete = complete_only
        write (u, *) "Cascade set:"
        write (u, "(3x,A)", advance="no") "Model:"
        if (associated (cascade_set%model)) then
            write (u, "(1x,A)") char (model_get_name (cascade_set%model))
        else
            write (u, "(1x,A)") "[none]"
        end if
    end if

```



```

write (u, "(3x,A)", advance="no") "n_in/out/tot ="
write (u, *) cascade_set%n_in, cascade_set%n_out, cascade_set%n_tot
write (u, "(3x,A)", advance="no") "depth_out/tot ="
write (u, *) cascade_set%depth_out, cascade_set%depth_tot
write (u, "(3x,A)", advance="no") "mass thr(s/t) ="
write (u, *) cascade_set%m_threshold_s, cascade_set%m_threshold_t
write (u, "(3x,A)", advance="no") "off shell ="
write (u, *) cascade_set%off_shell
write (u, "(3x,A)", advance="no") "keep_nonreson ="
write (u, *) cascade_set%keep_nonresonant
write (u, "(3x,A)", advance="no") "n_groves ="
write (u, *) cascade_set%n_groves
write (u, *)
write (u, *) "Cascade list:"
if (associated (cascade_set%first)) then
    cascade => cascade_set%first
    do while (associated (cascade))
        if (active .and. .not. cascade%active) cycle
        if (complete .and. .not. cascade%complete) cycle
        call cascade_write (cascade, unit)
        cascade => cascade%next
    end do
else
    write (u, *) "[empty]"
end if
write (u, *) "Hash array"
write (u, "(3x,A)", advance="no") "n_entries ="
write (u, *) cascade_set%n_entries
write (u, "(3x,A)", advance="no") "fill_ratio ="
write (u, *) cascade_set%fill_ratio
write (u, "(3x,A)", advance="no") "n_entries_max ="
write (u, *) cascade_set%n_entries_max
write (u, "(3x,A)", advance="no") "mask ="
write (u, *) cascade_set%mask
do i = 0, ubound (cascade_set%entry, 1)
    if (allocated (cascade_set%entry(i)%key)) then
        write (u, *) i
        call hash_entry_write (cascade_set%entry(i), u)
    end if
end do
end subroutine cascade_set_write

```

### 11.7.7 Adding cascades

Add a deep copy of a cascade to the set. The copy has all content of the original, but the pointers are nullified. We do not care whether insertion was successful or not. The pointer argument, if present, is assigned to the input cascade, or to the hash entry if it is already present.

The procedure is recursive: any daughter or mother entries are also deep-copied and added to the cascade set before the current copy is added.

*(Cascades: procedures)+≡*

```
recursive subroutine cascade_set_add_copy &
```

```

        (cascade_set, cascade_in, cascade_ptr)
type(cascade_set_t), intent(inout), target :: cascade_set
type(cascade_t), intent(in) :: cascade_in
type(cascade_t), optional, pointer :: cascade_ptr
type(cascade_t), pointer :: cascade
logical :: ok
allocate (cascade)
cascade = cascade_in
if (associated (cascade_in%daughter1)) call cascade_set_add_copy &
    (cascade_set, cascade_in%daughter1, cascade%daughter1)
if (associated (cascade_in%daughter2)) call cascade_set_add_copy &
    (cascade_set, cascade_in%daughter2, cascade%daughter2)
if (associated (cascade_in%mother)) call cascade_set_add_copy &
    (cascade_set, cascade_in%mother, cascade%mother)
cascade%next => null ()
call cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
end subroutine cascade_set_add_copy

```

Add a cascade to the set. This does not deep-copy. We first try to insert it in the hash array. If successful, add it to the list. Failure indicates that it is already present, and we drop it.

The hash key is built solely from the tree array, so neither particle codes nor resonances count, just topology.

Technically, hash and list receive only pointers, so the cascade can be considered as being in either of both. We treat it as part of the list.

*<Cascades: procedures>+≡*

```

subroutine cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
type(cascade_set_t), intent(inout), target :: cascade_set
type(cascade_t), intent(in), target :: cascade
logical, intent(out) :: ok
type(cascade_t), optional, pointer :: cascade_ptr
integer(i8), dimension(1) :: mold
call cascade_set_hash_insert &
    (cascade_set, transfer (cascade%tree, mold), cascade, ok, cascade_ptr)
if (ok) call cascade_set_list_add (cascade_set, cascade)
end subroutine cascade_set_add

```

Add a new cascade to the list:

*<Cascades: procedures>+≡*

```

subroutine cascade_set_list_add (cascade_set, cascade)
type(cascade_set_t), intent(inout) :: cascade_set
type(cascade_t), intent(in), target :: cascade
if (associated (cascade_set%last)) then
    cascade_set%last%next => cascade
else
    cascade_set%first => cascade
end if
cascade_set%last => cascade
end subroutine cascade_set_list_add

```

Add a cascade entry to the hash array:

*<Cascades: procedures>+≡*

```

subroutine cascade_set_hash_insert &
    (cascade_set, key, cascade, ok, cascade_ptr)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer(i8), dimension(:), intent(in) :: key
    type(cascade_t), intent(in), target :: cascade
    logical, intent(out) :: ok
    type(cascade_ptr), optional, pointer :: cascade_ptr
    integer(i32) :: h
    if (cascade_set%n_entries >= cascade_set%n_entries_max) &
        call cascade_set_hash_expand (cascade_set)
    h = hash (key)
    call cascade_set_hash_insert_rec &
        (cascade_set, h, h, key, cascade, ok, cascade_ptr)
end subroutine cascade_set_hash_insert

```

Double the hashtable size when necessary:

```

<Cascades: procedures>+≡
subroutine cascade_set_hash_expand (cascade_set)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(hash_entry_t), dimension(:), allocatable, target :: table_tmp
    type(cascade_p), pointer :: current
    integer :: i, s
    allocate (table_tmp (0:cascade_set%mask))
    table_tmp = cascade_set%entry
    deallocate (cascade_set%entry)
    s = 2 * size (table_tmp)
    cascade_set%n_entries = 0
    cascade_set%n_entries_max = s * cascade_set%fill_ratio
    cascade_set%mask = s - 1
    allocate (cascade_set%entry (0:cascade_set%mask))
    do i = 0, ubound (table_tmp, 1)
        current => table_tmp(i)%first
        do while (associated (current))
            call cascade_set_hash_insert_rec &
                (cascade_set, table_tmp(i)%hashval, table_tmp(i)%hashval, &
                    table_tmp(i)%key, current%cascade)
            current => current%next
        end do
    end do
end subroutine cascade_set_hash_expand

```

Insert the cascade at the bucket determined by the hash value. If the bucket is filled, check first for a collision (unequal keys). In that case, choose the following bucket and repeat. Otherwise, add the cascade to the bucket.

If the bucket is empty, record the hash value, allocate and store the key, and then add the cascade to the bucket.

If ok is present, before insertion we check whether the cascade is already stored, and return failure if yes.

```

<Cascades: procedures>+≡
recursive subroutine cascade_set_hash_insert_rec &
    (cascade_set, h, hashval, key, cascade, ok, cascade_ptr)
    type(cascade_set_t), intent(inout) :: cascade_set

```

```

integer(i32), intent(in) :: h, hashval
integer(i8), dimension(:), intent(in) :: key
type(cascade_t), intent(in), target :: cascade
logical, intent(out), optional :: ok
type(cascade_t), optional, pointer :: cascade_ptr
integer(i32) :: i
i = iand (h, cascade_set%mask)
if (allocated (cascade_set%entry(i)%key)) then
  if (size (cascade_set%entry(i)%key) /= size (key)) then
    call cascade_set_hash_insert_rec &
      (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
  else if (any (cascade_set%entry(i)%key /= key)) then
    call cascade_set_hash_insert_rec &
      (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
  else
    call hash_entry_add_cascade_ptr &
      (cascade_set%entry(i), cascade, ok, cascade_ptr)
  end if
else
  cascade_set%entry(i)%hashval = hashval
  allocate (cascade_set%entry(i)%key (size (key)))
  cascade_set%entry(i)%key = key
  call hash_entry_add_cascade_ptr &
    (cascade_set%entry(i), cascade, ok, cascade_ptr)
  cascade_set%n_entries = cascade_set%n_entries + 1
end if
end subroutine cascade_set_hash_insert_rec

```

### 11.7.8 External particles

We want to initialize the cascade set with the outgoing particles. In case of multiple processes, initial cascades are prepared for all of them. The hash array check ensures that no particle appears more than once at the same place.

*<Cascades: interfaces>+≡*

```

interface cascade_set_add_outgoing
  module procedure cascade_set_add_outgoing1
  module procedure cascade_set_add_outgoing2
end interface

```

*<Cascades: procedures>+≡*

```

subroutine cascade_set_add_outgoing2 (cascade_set, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(flavor_t), dimension(:,:), intent(in) :: flv
  integer :: pos, prc, n_out, n_prc
  type(cascade_t), pointer :: cascade
  logical :: ok
  n_out = size (flv, dim=1)
  n_prc = size (flv, dim=2)
  do prc = 1, n_prc
    do pos = 1, n_out
      allocate (cascade)
      call cascade_init_outgoing &

```

```

        (cascade, flv(pos,prc), pos, cascade_set%m_threshold_s)
    call cascade_set_add (cascade_set, cascade, ok)
    if (.not. ok) then
        deallocate (cascade)
    end if
end do
end do
end subroutine cascade_set_add_outgoing2

subroutine cascade_set_add_outgoing1 (cascade_set, flv)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(flavor_t), dimension(:), intent(in) :: flv
    integer :: pos, n_out
    type(cascade_t), pointer :: cascade
    logical :: ok
    n_out = size (flv, dim=1)
    do pos = 1, n_out
        allocate (cascade)
        call cascade_init_outgoing &
            (cascade, flv(pos), pos, cascade_set%m_threshold_s)
        call cascade_set_add (cascade_set, cascade, ok)
        if (.not. ok) then
            deallocate (cascade)
        end if
    end do
end subroutine cascade_set_add_outgoing1

```

The incoming particles are added one at a time. Nevertheless, we may have several processes which are looped over. At the first opportunity, we set the pointer `first_t` in the cascade set which should point to the first t-channel cascade.

Return the indices of the first and last cascade generated.

*<Cascades: interfaces>+≡*

```

interface cascade_set_add_incoming
    module procedure cascade_set_add_incoming0
    module procedure cascade_set_add_incoming1
end interface

```

*<Cascades: procedures>+≡*

```

subroutine cascade_set_add_incoming1 (cascade_set, n1, n2, pos, flv)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer, intent(out) :: n1, n2
    integer, intent(in) :: pos
    type(flavor_t), dimension(:), intent(in) :: flv
    integer :: prc, n_prc
    type(cascade_t), pointer :: cascade
    logical :: ok
    n1 = 0
    n2 = 0
    n_prc = size (flv)
    do prc = 1, n_prc
        allocate (cascade)
        call cascade_init_incoming &

```

```

        (cascade, flv(prc), pos, cascade_set%m_threshold_t)
    call cascade_set_add (cascade_set, cascade, ok)
    if (ok) then
        if (n1 == 0) n1 = cascade%index
        n2 = cascade%index
        if (.not. associated (cascade_set%first_t)) then
            cascade_set%first_t => cascade
        end if
    else
        deallocate (cascade)
    end if
end do
end subroutine cascade_set_add_incoming1

subroutine cascade_set_add_incoming0 (cascade_set, n1, n2, pos, flv)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer, intent(out) :: n1, n2
    integer, intent(in) :: pos
    type(flavor_t), intent(in) :: flv
    type(cascade_t), pointer :: cascade
    logical :: ok
    n1 = 0
    n2 = 0
    allocate (cascade)
    call cascade_init_incoming &
        (cascade, flv, pos, cascade_set%m_threshold_t)
    call cascade_set_add (cascade_set, cascade, ok)
    if (ok) then
        if (n1 == 0) n1 = cascade%index
        n2 = cascade%index
        if (.not. associated (cascade_set%first_t)) then
            cascade_set%first_t => cascade
        end if
    else
        deallocate (cascade)
    end if
end subroutine cascade_set_add_incoming0

```

### 11.7.9 Cascade combination I: flavor assignment

We have two disjunct cascades, now use the vertex table to determine the possible flavors of the combination cascade. For each possibility, try to generate a new cascade. The total cascade depth has to be one less than the limit, because this is reached by setting the keystone.

```

(Cascades: procedures) +=
subroutine cascade_match_pair (cascade_set, cascade1, cascade2, s_channel)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2
    logical, intent(in) :: s_channel
    integer, dimension(:), allocatable :: pdg3
    integer :: i, depth_max
    type(flavor_t) :: flv

```

```

if (s_channel) then
  depth_max = cascade_set%depth_out
else
  depth_max = cascade_set%depth_tot
end if
if (cascade1%depth + cascade2%depth < depth_max) then
  call model_match_vertex (cascade_set%model, &
    flavor_get_pdg (cascade1%flv), &
    flavor_get_pdg (cascade2%flv), &
    pdg3)
  do i = 1, size (pdg3)
    call flavor_init (flv, pdg3(i), cascade_set%model)
    if (s_channel) then
      call cascade_combine_s (cascade_set, cascade1, cascade2, flv)
    else
      call cascade_combine_t (cascade_set, cascade1, cascade2, flv)
    end if
  end do
  deallocate (pdg3)
end if
end subroutine cascade_match_pair

```

The triplet version takes a third cascade, and we check whether this triplet has a matching vertex in the database. If yes, we make a keystone cascade.

*(Cascades: procedures)* +=

```

subroutine cascade_match_triplet &
  (cascade_set, cascade1, cascade2, cascade3, s_channel)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
  logical, intent(in) :: s_channel
  integer :: depth_max
  depth_max = cascade_set%depth_tot
  if (cascade1%depth + cascade2%depth + cascade3%depth == depth_max) then
    if (model_check_vertex (cascade_set%model, &
      flavor_get_pdg (cascade1%flv), &
      flavor_get_pdg (cascade2%flv), &
      flavor_get_pdg (cascade3%flv))) then
      call cascade_combine_keystone &
        (cascade_set, cascade1, cascade2, cascade3, s_channel)
    end if
  end if
end subroutine cascade_match_triplet

```

### 11.7.10 Cascade combination II: kinematics setup and check

Having three matching flavors, we start constructing the combination cascade. We look at the mass hierarchies and determine whether the cascade is to be kept. In passing we set mapping modes, resonance properties and such.

If successful, the cascade is finalized. For a resonant cascade, we prepare in addition a copy without the resonance.

*(Cascades: procedures)* +=

```

subroutine cascade_combine_s (cascade_set, cascade1, cascade2, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2
  type(flavor_t), intent(in) :: flv
  type(cascade_t), pointer :: cascade3, cascade4
  logical :: keep
  keep = .false.
  allocate (cascade3)
  call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
  cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
  cascade3%flv = flavor_anti (flv)
  cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
  cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
  cascade3%m_min = cascade1%m_min + cascade2%m_min
  cascade3%m_rea = flavor_get_mass (flv)
  if (cascade3%m_rea > cascade_set%m_threshold_s) then
    cascade3%m_eff = cascade3%m_rea
  end if
  ! Potentially resonant cases [sqrts = m_rea for on-shell decay]
  if (cascade3%m_rea > cascade3%m_min &
    .and. cascade3%m_rea <= cascade_set%sqrts) then
    if (flavor_get_width (flv) /= 0) then
      if (cascade1%on_shell .or. cascade2%on_shell) then
        keep = .true.
        cascade3%mapping = S_CHANNEL
        cascade3%resonant = .true.
      end if
    else
      call warn_decay (flv)
    end if
  ! Collinear and IR singular cases
  else if (cascade3%m_rea < cascade_set%sqrts) then
    ! Massless splitting
    if (cascade1%m_eff == 0 .and. cascade2%m_eff == 0 &
      .and. cascade3%depth <= 3) then
      keep = .true.
      cascade3%log_enhanced = .true.
      if (cascade3%is_vector) then
        if (cascade1%is_vector .and. cascade2%is_vector) then
          cascade3%mapping = COLLINEAR ! three-vector-vertex
        else
          cascade3%mapping = INFRARED ! vector splitting into matter
        end if
      else
        if (cascade1%is_vector .or. cascade2%is_vector) then
          cascade3%mapping = COLLINEAR ! vector radiation off matter
        else
          cascade3%mapping = INFRARED ! scalar radiation/splitting
        end if
      end if
    ! IR radiation off massive particle
    else if (cascade3%m_eff > 0 .and. cascade1%m_eff > 0 &
      .and. cascade2%m_eff == 0 &
      .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &

```



```

        .and. abs (cascade3%m_eff - cascade1%m_eff) &
            < cascade_set%m_threshold_s) &
        then
            keep = .true.
            cascade3%log_enhanced = .true.
            cascade3%mapping = RADIATION
        else if (cascade3%m_eff > 0 .and. cascade2%m_eff > 0 &
            .and. cascade1%m_eff == 0 &
            .and. (cascade2%on_shell .or. cascade2%mapping == RADIATION) &
            .and. abs (cascade3%m_eff - cascade2%m_eff) &
                < cascade_set%m_threshold_s) &
            then
                keep = .true.
                cascade3%log_enhanced = .true.
                cascade3%mapping = RADIATION
            end if
        end if
        ! Non-singular cases, including failed resonances
        if (.not. keep) then
            ! Two on-shell particles from a virtual mother
            if (cascade1%on_shell .or. cascade2%on_shell) then
                keep = .true.
                cascade3%m_eff = max (cascade3%m_min, &
                    cascade1%m_eff + cascade2%m_eff)
                if (cascade3%m_eff < cascade_set%m_threshold_s) then
                    cascade3%m_eff = 0
                end if
            end if
        end if
        ! Complete and register the cascade (two in case of resonance)
        if (keep) then
            cascade3%on_shell = cascade3%resonant .or. cascade3%log_enhanced
            if (cascade3%resonant) then
                cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
                if (cascade_set%keep_nonresonant) then
                    allocate (cascade4)
                    cascade4 = cascade3
                    cascade4%index = cascade_index ()
                    cascade4%pdg = UNDEFINED
                    cascade4%mapping = NO_MAPPING
                    cascade4%resonant = .false.
                    cascade4%on_shell = .false.
                end if
                cascade3%m_min = cascade3%m_rea
                call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
                if (cascade_set%keep_nonresonant) then
                    call cascade_fusion (cascade_set, cascade1, cascade2, cascade4)
                end if
            else
                call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
            end if
        else
            deallocate (cascade3)
        end if

```

```

contains
  subroutine warn_decay (flv)
    type(flavor_t), intent(in) :: flv
    integer :: i
    integer, dimension(MAX_WARN_RESONANCE), save :: warned_code = 0
    LOOP_WARNED: do i = 1, MAX_WARN_RESONANCE
      if (warned_code(i) == 0) then
        warned_code(i) = flavor_get_pdg (flv)
        write (msg_buffer, "(A)") &
          & " Intermediate decay of zero-width particle " &
          & // char (flavor_get_name (flv)) &
          & // " may be possible."
        call msg_warning
        exit LOOP_WARNED
      else if (warned_code(i) == flavor_get_pdg (flv)) then
        exit LOOP_WARNED
      end if
    end do LOOP_WARNED
  end subroutine warn_decay
end subroutine cascade_combine_s

```

*(Limits: public parameters)+≡*

```
integer, parameter, public :: MAX_WARN_RESONANCE = 50
```

This is the t-channel version. `cascade1` is t-channel and contains the seed, `cascade2` is s-channel. We check for kinematically allowed beam decay (which is a fatal error), or massless splitting / soft radiation. The cascade is kept in all remaining cases and submitted for registration.

*(Cascades: procedures)+≡*

```

subroutine cascade_combine_t (cascade_set, cascade1, cascade2, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2
  type(flavor_t), intent(in) :: flv
  type(cascade_t), pointer :: cascade3
  allocate (cascade3)
  call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
  cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
  cascade3%flv = flavor_anti (flv)
  cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
  cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
  if (cascade1%incoming) then
    cascade3%m_min = cascade2%m_min
  else
    cascade3%m_min = cascade1%m_min + cascade2%m_min
  end if
  cascade3%m_rea = flavor_get_mass (flv)
  if (cascade3%m_rea > cascade_set%m_threshold_t) then
    cascade3%m_eff = max (cascade3%m_rea, cascade2%m_eff)
  else if (cascade2%m_eff > cascade_set%m_threshold_t) then
    cascade3%m_eff = cascade2%m_eff
  else
    cascade3%m_eff = 0
  end if
  ! Allowed decay of beam particle

```

```

if (cascade1%incoming &
    .and. cascade1%m_rea > cascade2%m_rea + cascade3%m_rea) then
    call beam_decay (cascade_set%fatal_beam_decay)
! Massless splitting
else if (cascade1%m_eff == 0 &
    .and. cascade2%m_eff < cascade_set%m_threshold_t &
    .and. cascade3%m_eff == 0) then
    cascade3%mapping = U_CHANNEL
    cascade3%log_enhanced = .true.
! IR radiation off massive particle
else if (cascade1%m_eff /= 0 .and. cascade2%m_eff == 0 &
    .and. cascade3%m_eff /= 0 &
    .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
    .and. abs (cascade1%m_eff - cascade3%m_eff) &
        < cascade_set%m_threshold_t) &
    then
        cascade3%pdg = abs (flavor_get_pdg (flv))
        cascade3%log_enhanced = .true.
        cascade3%mapping = RADIATION
    end if
    cascade3%t_channel = .true.
    call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
contains
subroutine beam_decay (fatal_beam_decay)
    logical, intent(in) :: fatal_beam_decay
    write (msg_buffer, "(1x,A,1x,'->',1x,A,1x,A)") &
        char (flavor_get_name (cascade1%flv)), &
        char (flavor_get_name (cascade3%flv)), &
        char (flavor_get_name (cascade2%flv))
    call msg_message
    write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
        char (flavor_get_name (cascade1%flv)), cascade1%m_rea
    call msg_message
    write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
        char (flavor_get_name (cascade3%flv)), cascade3%m_rea
    call msg_message
    write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
        char (flavor_get_name (cascade2%flv)), cascade2%m_rea
    call msg_message
    if (fatal_beam_decay) then
        call msg_fatal (" Phase space: Initial beam particle can decay")
    else
        call msg_warning (" Phase space: Initial beam particle can decay")
    end if
end subroutine beam_decay
end subroutine cascade_combine_t

```

Here we complete a decay cascade. The third input is the single-particle cascade for the initial particle. There is no resonance or mapping assignment. The only condition for keeping the cascade is the mass sum of the final state, which must be less than the available energy.

Two modifications are necessary for scattering cascades: a pure s-channel diagram (cascade1 is the incoming particle) do not have a logarithmic mapping

at top-level. And in a t-channel diagram, the last line exchanged is mapped t-channel, not u-channel. In both cases we register a new cascade with the modified mapping.

```

(Cascades: procedures) +=
  subroutine cascade_combine_keystone &
    (cascade_set, cascade1, cascade2, cascade3, s_channel)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
    logical, intent(in) :: s_channel
    type(cascade_t), pointer :: cascade4, cascade0
    logical :: keep, ok
    keep = .false.
    allocate (cascade4)
    call cascade_init &
      (cascade4, cascade1%depth + cascade2%depth + cascade3%depth)
    cascade4%complete = .true.
!    cascade4%bincode = ior (ior (cascade1%bincode, cascade2%bincode), &
!      cascade3%bincode)
    if (s_channel) then
      cascade4%bincode = ior (cascade1%bincode, cascade2%bincode)
    else
      cascade4%bincode = cascade3%bincode
    end if
    cascade4%flv = cascade3%flv
    cascade4%pdg = cascade3%pdg
    cascade4%mapping = EXTERNAL_PRT
    cascade4%is_vector = cascade3%is_vector
    cascade4%m_min = cascade1%m_min + cascade2%m_min
    cascade4%m_rea = cascade3%m_rea
    cascade4%m_eff = cascade3%m_rea
    if (cascade4%m_min < cascade_set%sqrts) then
      keep = .true.
    end if
    if (keep) then
      if (cascade1%incoming .and. cascade2%log_enhanced) then
        allocate (cascade0)
        cascade0 = cascade2
        cascade0%next => null ()
        cascade0%index = cascade_index ()
        cascade0%mapping = NO_MAPPING
        cascade0%log_enhanced = .false.
        cascade0%n_log_enhanced = cascade0%n_log_enhanced - 1
        cascade0%tree_mapping(cascade0%depth) = NO_MAPPING
        call cascade_keystone &
          (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
        if (ok) call cascade_set_add (cascade_set, cascade0, ok)
      else if (cascade1%t_channel .and. cascade1%mapping == U_CHANNEL) then
        allocate (cascade0)
        cascade0 = cascade1
        cascade0%next => null ()
        cascade0%index = cascade_index ()
        cascade0%mapping = T_CHANNEL
        cascade0%tree_mapping(cascade0%depth) = T_CHANNEL
        call cascade_keystone &

```

```

        (cascade_set, cascade0, cascade2, cascade3, cascade4, ok)
    if (ok) call cascade_set_add (cascade_set, cascade0, ok)
    else
        call cascade_keystone &
            (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
    end if
else
    deallocate (cascade4)
end if
end subroutine cascade_combine_keystone

```

### 11.7.11 Cascade combination III: node connections and tree fusion

Here we assign global tree properties. If the allowed number of off-shell lines is exceeded, discard the new cascade. Otherwise, assign the trees, sort them, and assign connections. Finally, append the cascade to the list. This may fail (because in the hash array there is already an equivalent cascade). On failure, discard the cascade.

```

(Cascades: procedures) +=
subroutine cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2
    type(cascade_t), pointer :: cascade3
    integer :: i1, i2, i3, i4
    logical :: ok
    cascade3%internal = (cascade3%depth - 3) / 2
    if (cascade3%resonant) then
        cascade3%multiplicity = 1
        cascade3%n_resonances = &
            cascade1%n_resonances + cascade2%n_resonances + 1
    else
        cascade3%multiplicity = cascade1%multiplicity + cascade2%multiplicity
        cascade3%n_resonances = cascade1%n_resonances + cascade2%n_resonances
    end if
    if (cascade3%log_enhanced) then
        cascade3%n_log_enhanced = &
            cascade1%n_log_enhanced + cascade2%n_log_enhanced + 1
    else
        cascade3%n_log_enhanced = &
            cascade1%n_log_enhanced + cascade2%n_log_enhanced
    end if
    if (cascade3%resonant) then
        cascade3%n_off_shell = 0
    else if (cascade3%log_enhanced) then
        cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
    else
        cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell + 1
    end if
    if (cascade3%t_channel) then
        cascade3%n_t_channel = cascade1%n_t_channel + 1
    end if
end if

```

```

if (cascade3%n_off_shell > cascade_set%off_shell) then
  deallocate (cascade3)
else if (cascade3%n_t_channel > cascade_set%t_channel) then
  deallocate (cascade3)
else
  i1 = cascade1%depth
  i2 = i1 + 1
  i3 = i1 + cascade2%depth
  i4 = cascade3%depth
  cascade3%tree(:i1) = cascade1%tree
  where (cascade1%tree_mapping > NO_MAPPING)
    cascade3%tree_pdg(:i1) = cascade1%tree_pdg
  elsewhere
    cascade3%tree_pdg(:i1) = UNDEFINED
  end where
  cascade3%tree_mapping(:i1) = cascade1%tree_mapping
  cascade3%tree_resonant(:i1) = cascade1%tree_resonant
  cascade3%tree(i2:i3) = cascade2%tree
  where (cascade2%tree_mapping > NO_MAPPING)
    cascade3%tree_pdg(i2:i3) = cascade2%tree_pdg
  elsewhere
    cascade3%tree_pdg(i2:i3) = UNDEFINED
  end where
  cascade3%tree_mapping(i2:i3) = cascade2%tree_mapping
  cascade3%tree_resonant(i2:i3) = cascade2%tree_resonant
  cascade3%tree(i4) = cascade3%bincode
  cascade3%tree_pdg(i4) = cascade3%pdg
  cascade3%tree_mapping(i4) = cascade3%mapping
  cascade3%tree_resonant(i4) = cascade3%resonant
  call tree_sort (cascade3%tree, &
    cascade3%tree_pdg, cascade3%tree_mapping, cascade3%tree_resonant)
  cascade3%has_children = .true.
  cascade3%daughter1 => cascade1
  cascade3%daughter2 => cascade2
  call cascade_set_add (cascade_set, cascade3, ok)
  if (.not. ok) deallocate (cascade3)
end if
end subroutine cascade_fusion

```

Here we combine a cascade pair with an incoming particle, i.e., we set a keystone. Otherwise, this is similar. On the first opportunity, we set the **first\_k** pointer in the cascade set.

*(Cascades: procedures)* +=

```

subroutine cascade_keystone &
  (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
  type(cascade_t), pointer :: cascade4
  logical, intent(out) :: ok
  integer :: i1, i2, i3, i4
  cascade4%internal = (cascade4%depth - 3) / 2
  cascade4%multiplicity = cascade1%multiplicity + cascade2%multiplicity
  cascade4%n_resonances = cascade1%n_resonances + cascade2%n_resonances

```

```

cascade4%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
cascade4%n_log_enhanced = &
    cascade1%n_log_enhanced + cascade2%n_log_enhanced
cascade4%n_t_channel = cascade1%n_t_channel + cascade2%n_t_channel
if (cascade4%n_off_shell > cascade_set%off_shell) then
    deallocate (cascade4)
    ok = .false.
else if (cascade4%n_t_channel > cascade_set%t_channel) then
    deallocate (cascade4)
    ok = .false.
else
    i1 = cascade1%depth
    i2 = i1 + 1
    i3 = i1 + cascade2%depth
    i4 = cascade4%depth
    cascade4%tree(:i1) = cascade1%tree
    where (cascade1%tree_mapping > NO_MAPPING)
        cascade4%tree_pdg(:i1) = cascade1%tree_pdg
    elsewhere
        cascade4%tree_pdg(:i1) = UNDEFINED
    end where
    cascade4%tree_mapping(:i1) = cascade1%tree_mapping
    cascade4%tree_resonant(:i1) = cascade1%tree_resonant
    cascade4%tree(i2:i3) = cascade2%tree
    where (cascade2%tree_mapping > NO_MAPPING)
        cascade4%tree_pdg(i2:i3) = cascade2%tree_pdg
    elsewhere
        cascade4%tree_pdg(i2:i3) = UNDEFINED
    end where
    cascade4%tree_mapping(i2:i3) = cascade2%tree_mapping
    cascade4%tree_resonant(i2:i3) = cascade2%tree_resonant
!    cascade4%tree(i4) = cascade3%bincode
    cascade4%tree(i4) = cascade4%bincode
    cascade4%tree_pdg(i4) = UNDEFINED
    cascade4%tree_mapping(i4) = cascade4%mapping
    cascade4%tree_resonant(i4) = .false.
    call tree_sort (cascade4%tree, &
        cascade4%tree_pdg, cascade4%tree_mapping, cascade4%tree_resonant)
    cascade4%has_children = .true.
    cascade4%daughter1 => cascade1
    cascade4%daughter2 => cascade2
    cascade4%mother => cascade3
    call cascade_set_add (cascade_set, cascade4, ok)
    if (ok) then
        if (.not. associated (cascade_set%first_k)) then
            cascade_set%first_k => cascade4
        end if
    else
        deallocate (cascade4)
    end if
end if
end subroutine cascade_keystone

```

Sort a tree (array of binary codes) and particle code array simultaneously, by

ascending binary codes. A convenient method is to use the `maxloc` function iteratively, to find and remove the largest entry in the tree array one by one.

*(Cascades: procedures)*+≡

```

subroutine tree_sort (tree, pdg, mapping, resonant)
  integer(TC), dimension(:), intent(inout) :: tree
  integer, dimension(:), intent(inout) :: pdg, mapping
  logical, dimension(:), intent(inout) :: resonant
  integer(TC), dimension(size(tree)) :: tree_tmp
  integer, dimension(size(pdg)) :: pdg_tmp, mapping_tmp
  logical, dimension(size(resonant)) :: resonant_tmp
  integer, dimension(1) :: pos
  integer :: i
  tree_tmp = tree
  pdg_tmp = pdg
  mapping_tmp = mapping
  resonant_tmp = resonant
  do i = size(tree), 1, -1
    pos = maxloc (tree_tmp)
    tree(i) = tree_tmp (pos(1))
    pdg(i) = pdg_tmp (pos(1))
    mapping(i) = mapping_tmp (pos(1))
    resonant(i) = resonant_tmp (pos(1))
    tree_tmp(pos(1)) = 0
  end do
end subroutine tree_sort

```

### 11.7.12 Cascade set generation

These procedures loop over cascades and build up the cascade set. After each iteration of the innermost loop, we set a breakpoint.

s-channel: We use a nested scan to combine all cascades with all other cascades.

*(Cascades: procedures)*+≡

```

subroutine cascade_set_generate_s (cascade_set)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), pointer :: cascade1, cascade2
  cascade1 => cascade_set%first
  LOOP1: do while (associated (cascade1))
    cascade2 => cascade_set%first
    LOOP2: do while (associated (cascade2))
      if (cascade2%index >= cascade1%index) exit LOOP2
      if (cascade1 .disjunct. cascade2) then
        call cascade_match_pair (cascade_set, cascade1, cascade2, .true.)
      end if
      call terminate_now_if_signal ()
      cascade2 => cascade2%next
    end do LOOP2
    cascade1 => cascade1%next
  end do LOOP1
end subroutine cascade_set_generate_s

```



The t-channel cascades are directed and have a seed (one of the incoming particles) and a target (the other one). We loop over all possible seeds and targets. Inside this, we loop over all t-channel cascades (`cascade1`) and s-channel cascades (`cascade2`) and try to combine them.

```

(Cascades: procedures) +=
  subroutine cascade_set_generate_t (cascade_set, pos_seed, pos_target)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer, intent(in) :: pos_seed, pos_target
    type(cascade_t), pointer :: cascade_seed, cascade_target
    type(cascade_t), pointer :: cascade1, cascade2
    integer(TC) :: bc_seed, bc_target
    bc_seed = ibset (0_TC, pos_seed-1)
    bc_target = ibset (0_TC, pos_target-1)
    cascade_seed => cascade_set%first_t
    LOOP_SEED: do while (associated (cascade_seed))
      if (cascade_seed%bincode == bc_seed) then
        cascade_target => cascade_set%first_t
        LOOP_TARGET: do while (associated (cascade_target))
          if (cascade_target%bincode == bc_target) then
            cascade1 => cascade_set%first_t
            LOOP_T: do while (associated (cascade1))
              if ((cascade1 .disjunct. cascade_target) &
                  .and. .not. (cascade1 .disjunct. cascade_seed)) then
                cascade2 => cascade_set%first
                LOOP_S: do while (associated (cascade2))
                  if ((cascade2 .disjunct. cascade_target) &
                      .and. (cascade2 .disjunct. cascade1)) then
                    call cascade_match_pair &
                      (cascade_set, cascade1, cascade2, .false.)
                  end if
                  call terminate_now_if_signal ()
                  cascade2 => cascade2%next
                end do LOOP_S
              end if
              call terminate_now_if_signal ()
              cascade1 => cascade1%next
            end do LOOP_T
          end if
          call terminate_now_if_signal ()
          cascade_target => cascade_target%next
        end do LOOP_TARGET
      end if
      cascade_seed => cascade_seed%next
    end do LOOP_SEED
  end subroutine cascade_set_generate_t

```

This part completes the phase space for decay processes. It is similar to s-channel cascade generation, but combines two cascade with the particular cascade of the incoming particle. This particular cascade is expected to be pointed at by `first_t`.

```

(Cascades: procedures) +=
  subroutine cascade_set_generate_decay (cascade_set)

```

```

type(cascade_set_t), intent(inout), target :: cascade_set
type(cascade_t), pointer :: cascade1, cascade2
type(cascade_t), pointer :: cascade_in
cascade_in => cascade_set%first_t
cascade1 => cascade_set%first
do while (associated (cascade1))
  if (cascade1 .disjunct. cascade_in) then
    cascade2 => cascade1%next
    do while (associated (cascade2))
      if ((cascade2 .disjunct. cascade1) &
          .and. (cascade2 .disjunct. cascade_in)) then
        call cascade_match_triplet (cascade_set, &
                                     cascade1, cascade2, cascade_in, .true.)
      end if
      call terminate_now_if_signal ()
      cascade2 => cascade2%next
    end do
  end if
  call terminate_now_if_signal ()
  cascade1 => cascade1%next
end do
end subroutine cascade_set_generate_decay

```

This part completes the phase space for scattering processes. We combine a t-channel cascade (containing the seed) with a s-channel cascade and the target.

*(Cascades: procedures)* +=

```

subroutine cascade_set_generate_scattering &
  (cascade_set, ns1, ns2, nt1, nt2, pos_seed, pos_target)
type(cascade_set_t), intent(inout), target :: cascade_set
integer, intent(in) :: pos_seed, pos_target
integer, intent(in) :: ns1, ns2, nt1, nt2
type(cascade_t), pointer :: cascade_seed, cascade_target
type(cascade_t), pointer :: cascade1, cascade2
integer(TC) :: bc_seed, bc_target
bc_seed = ibset (0_TC, pos_seed-1)
bc_target = ibset (0_TC, pos_target-1)
cascade_seed => cascade_set%first_t
LOOP_SEED: do while (associated (cascade_seed))
  if (cascade_seed%index < ns1) then
    cascade_seed => cascade_seed%next
    cycle LOOP_SEED
  else if (cascade_seed%index > ns2) then
    exit LOOP_SEED
  else if (cascade_seed%bincode == bc_seed) then
    cascade_target => cascade_set%first_t
    LOOP_TARGET: do while (associated (cascade_target))
      if (cascade_target%index < nt1) then
        cascade_target => cascade_target%next
        cycle LOOP_TARGET
      else if (cascade_target%index > nt2) then
        exit LOOP_TARGET
      else if (cascade_target%bincode == bc_target) then
        cascade1 => cascade_set%first_t

```

```

      LOOP_T: do while (associated (cascade1))
        if ((cascade1 .disjunct. cascade_target) &
            .and. .not. (cascade1 .disjunct. cascade_seed)) then
          cascade2 => cascade_set%first
          LOOP_S: do while (associated (cascade2))
            if ((cascade2 .disjunct. cascade_target) &
                .and. (cascade2 .disjunct. cascade1)) then
              call cascade_match_triplet (cascade_set, &
                                          cascade1, cascade2, cascade_target, .false.)
            end if
            call terminate_now_if_signal ()
            cascade2 => cascade2%next
          end do LOOP_S
        end if
        call terminate_now_if_signal ()
        cascade1 => cascade1%next
      end do LOOP_T
    end if
    call terminate_now_if_signal ()
    cascade_target => cascade_target%next
  end do LOOP_TARGET
end if
call terminate_now_if_signal ()
cascade_seed => cascade_seed%next
end do LOOP_SEED
end subroutine cascade_set_generate_scattering

```

### 11.7.13 Groves

Before assigning groves, assign hashcodes to the resonance patterns, so they can easily be compared.

*(Cascades: procedures)*+≡

```

subroutine cascade_set_assign_resonance_hash (cascade_set)
  type(cascade_set_t), intent(inout) :: cascade_set
  type(cascade_t), pointer :: cascade
  cascade => cascade_set%first_k
  do while (associated (cascade))
    call cascade_assign_resonance_hash (cascade)
    cascade => cascade%next
  end do
end subroutine cascade_set_assign_resonance_hash

```

After all cascades are recorded, we group the complete cascades in groves. A grove consists of cascades with identical multiplicity, number of resonances, log-enhanced, t-channel lines, and resonance flavors.

*(Cascades: procedures)*+≡

```

subroutine cascade_set_assign_groves (cascade_set)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), pointer :: cascade1, cascade2
  integer :: multiplicity
  integer :: n_resonances, n_log_enhanced, n_t_channel, n_off_shell
  integer :: res_hash

```

```

integer :: grove
grove = 0
cascade1 => cascade_set%first_k
do while (associated (cascade1))
  if (cascade1%active .and. cascade1%complete &
      .and. cascade1%grove == 0) then
    grove = grove + 1
    cascade1%grove = grove
    multiplicity = cascade1%multiplicity
    n_resonances = cascade1%n_resonances
    n_log_enhanced = cascade1%n_log_enhanced
    n_off_shell = cascade1%n_off_shell
    n_t_channel = cascade1%n_t_channel
    res_hash = cascade1%res_hash
    cascade2 => cascade1%next
    do while (associated (cascade2))
      if (cascade2%grove == 0) then
        if (cascade2%multiplicity == multiplicity &
            .and. cascade2%n_resonances == n_resonances &
            .and. cascade2%n_log_enhanced == n_log_enhanced &
            .and. cascade2%n_off_shell == n_off_shell &
            .and. cascade2%n_t_channel == n_t_channel &
            .and. cascade2%res_hash == res_hash) then
          cascade2%grove = grove
        end if
      end if
      call terminate_now_if_signal ()
      cascade2 => cascade2%next
    end do
  end if
  call terminate_now_if_signal ()
  cascade1 => cascade1%next
end do
cascade_set%n_groves = grove
end subroutine cascade_set_assign_groves

```

### 11.7.14 Generate the phase space file

Generate a complete phase space configuration.

For each flavor assignment: First, all s-channel graphs that can be built up from the outgoing particles. Then we distinguish (1) decay, where we complete the s-channel graphs by connecting to the input line, and (2) scattering, where we now generate t-channel graphs by introducing an incoming particle, and complete this by connecting to the other incoming particle.

After all cascade sets have been generated, merge them into a common set. This eliminates redundancies between flavor assignments.

```

<Cascades: public>+=
  public :: cascade_set_generate

<Cascades: procedures>+=
  subroutine cascade_set_generate &
    (cascade_set, model, n_in, n_out, flv, phs_par, fatal_beam_decay)

```

```

type(cascade_set_t), intent(out) :: cascade_set
type(model_t), intent(in), target :: model
integer, intent(in) :: n_in, n_out
type(flavor_t), dimension(:,:), intent(in) :: flv
type(phs_parameters_t), intent(in) :: phs_par
logical, intent(in) :: fatal_beam_decay
type(cascade_set_t), dimension(:), allocatable :: cset
type(cascade_t), pointer :: cascade
integer :: i
if (phase_space_vanishes (phs_par%sqrts, n_in, flv)) return
call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
    fatal_beam_decay, flv)
allocate (cset (size (flv, 2)))
do i = 1, size (cset)
    call cascade_set_generate_single (cset(i), &
        model, n_in, n_out, flv(:,i), phs_par, fatal_beam_decay)
    cascade => cset(i)%first_k
    do while (associated (cascade))
        if (cascade%active .and. cascade%complete) then
            call cascade_set_add_copy (cascade_set, cascade)
        end if
        cascade => cascade%next
    end do
    call cascade_set_final (cset(i))
end do
cascade_set%first_k => cascade_set%first
call cascade_set_assign_resonance_hash (cascade_set)
call cascade_set_assign_groves (cascade_set)
end subroutine cascade_set_generate

```

This generates phase space for a single channel, without assigning groves.

*(Cascades: procedures)*+≡

```

subroutine cascade_set_generate_single (cascade_set, &
    model, n_in, n_out, flv, phs_par, fatal_beam_decay)
type(cascade_set_t), intent(out) :: cascade_set
type(model_t), intent(in), target :: model
integer, intent(in) :: n_in, n_out
type(flavor_t), dimension(:), intent(in) :: flv
type(phs_parameters_t), intent(in) :: phs_par
logical, intent(in) :: fatal_beam_decay
integer :: n11, n12, n21, n22
call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
    fatal_beam_decay)
call cascade_set_add_outgoing (cascade_set, flv(n_in+1:))
call cascade_set_generate_s (cascade_set)
select case (n_in)
case(1)
    call cascade_set_add_incoming &
        (cascade_set, n11, n12, n_out + 1, flv(1))
    call cascade_set_generate_decay (cascade_set)
case(2)
    call cascade_set_add_incoming &
        (cascade_set, n11, n12, n_out + 1, flv(2))

```

```

        call cascade_set_add_incoming &
            (cascade_set, n21, n22, n_out + 2, flv(1))
        call cascade_set_generate_t (cascade_set, n_out + 1, n_out + 2)
        call cascade_set_generate_t (cascade_set, n_out + 2, n_out + 1)
        call cascade_set_generate_scattering &
            (cascade_set, n11, n12, n21, n22, n_out + 1, n_out + 2)
        call cascade_set_generate_scattering &
            (cascade_set, n21, n22, n11, n12, n_out + 2, n_out + 1)
    end select
end subroutine cascade_set_generate_single

```

Sanity check: Before anything else is done, check if there could possibly be any phase space.

*<Cascades: procedures>+≡*

```

function phase_space_vanishes (sqrts, n_in, flv) result (flag)
    logical :: flag
    real(default), intent(in) :: sqrts
    integer, intent(in) :: n_in
    type(flavor_t), dimension(:,:), intent(in) :: flv
    real(default), dimension(:,:), allocatable :: mass
    real(default), dimension(:), allocatable :: mass_in, mass_out
    integer :: n_prt, n_flv
    flag = .false.
    if (sqrts <= 0) then
        call msg_error ("Phase space vanishes (sqrts must be positive)")
        flag = .true.; return
    end if
    n_prt = size (flv, 1)
    n_flv = size (flv, 2)
    allocate (mass (n_prt, n_flv), mass_in (n_flv), mass_out (n_flv))
    mass = flavor_get_mass (flv)
    mass_in = sum (mass(:n_in,:), 1)
    mass_out = sum (mass(n_in+1:,:), 1)
    if (any (mass_in > sqrts)) then
        call msg_error ("Mass sum of incoming particles " &
            // "is more than available energy")
        flag = .true.; return
    end if
    if (any (mass_out > sqrts)) then
        call msg_error ("Mass sum of outgoing particles " &
            // "is more than available energy")
        flag = .true.; return
    end if
end function phase_space_vanishes

```

### 11.7.15 Test

*<Cascades: public>+≡*

```
public :: cascade_test
```

*<Cascades: procedures>+≡*

```

subroutine cascade_test
    use os_interface, only: os_data_t

```

```

type(os_data_t) :: os_data
type(model_t), pointer :: model
type(flavor_t), dimension(5,2) :: flv
type(cascade_set_t) :: cascade_set
type(string_t) :: name, filename
type(phs_parameters_t) :: phs_par
name = "SM"
filename = "SM.mdl"
call syntax_model_file_init ()
call model_list_read_model (name, filename, os_data, model)
call model_write (model, verbose=.true.)
call flavor_init (flv(1,1), 2, model)
call flavor_init (flv(2,1),-2, model)
call flavor_init (flv(3,1), 1, model)
call flavor_init (flv(4,1),-1, model)
call flavor_init (flv(5,1),21, model)
call flavor_init (flv(1,2), 2, model)
call flavor_init (flv(2,2),-2, model)
call flavor_init (flv(3,2), 2, model)
call flavor_init (flv(4,2),-2, model)
call flavor_init (flv(5,2),21, model)
phs_par%sqrts = 1000._default
phs_par%off_shell = 2
call cascade_set_generate (cascade_set, model, 2, 3, flv, phs_par,.true.)
call cascade_set_write (cascade_set)
call cascade_set_write_file_format (cascade_set)
call cascade_set_final (cascade_set)
call model_list_final ()
end subroutine cascade_test

```

## 11.8 WOOD phase space

This is the module that interfaces the `phs_forests` phase-space treatment and the `cascades` module for generating phase-space channels. As an extension of the `phs_base` abstract type, the phase-space configuration and instance implement the standard API.

(Currently, this is the only generic phase-space implementation of WHIZARD. For trivial two-particle phase space, there is `phs_wood` as an alternative.)

`<phs_wood.f90>`≡  
*<File header>*

```

module phs_wood

  <Use kinds>
  <Use strings>
  <Use file utils>
  use diagnostics !NODEP!
  use unit_tests
  use os_interface
  use md5
  use constants !NODEP!

```

```

use lorentz !NODEP!
use variables
use models
use flavors
use process_constants
use sf_mappings
use sf_base
use phs_base
use mappings
use phs_forests
use cascades

```

*⟨Standard module head⟩*

*⟨PHS wood: public⟩*

*⟨PHS wood: types⟩*

contains

*⟨PHS wood: procedures⟩*

*⟨PHS wood: tests⟩*

end module phs\_wood

### 11.8.1 Configuration

*⟨PHS wood: public⟩*≡

```
public :: phs_wood_config_t
```

*⟨PHS wood: types⟩*≡

```

type, extends (phs_config_t) :: phs_wood_config_t
  character(32) :: md5sum_forest = ""
  integer :: io_unit = 0
  logical :: io_unit_keep_open = .false.
  logical :: use_equivalences = .false.
  type(mapping_defaults_t) :: mapping_defaults
  type(phs_parameters_t) :: par
  type(cascade_set_t), allocatable :: cascade_set
  type(phs_forest_t) :: forest
contains
  ⟨PHS wood: phs wood config: TBP⟩
end type phs_wood_config_t

```

Finalizer. We should delete the cascade set and the forest subobject.

Also close the I/O unit, just in case. (We assume that `io_unit` is not standard input/output.)

*⟨PHS wood: phs wood config: TBP⟩*≡

```
procedure :: final => phs_wood_config_final
```

*⟨PHS wood: procedures⟩*≡

```

subroutine phs_wood_config_final (object)
  class(phs_wood_config_t), intent(inout) :: object

```



```

logical :: opened
if (object%io_unit /= 0) then
  inquire (unit = object%io_unit, opened = opened)
  if (opened) close (object%io_unit)
end if
call object%clear_phase_space ()
call phs_forest_final (object%forest)
end subroutine phs_wood_config_final

```

Output. The contents of the PHS forest are not printed explicitly.

```

<PHS wood: phs wood config: TBP>+≡
  procedure :: write => phs_wood_config_write
<PHS wood: procedures>+≡
  subroutine phs_wood_config_write (object, unit)
    class(phs_wood_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") &
      "Partonic phase-space configuration (phase-space forest):"
    call object%base_write (unit)
    ! if (object%md5sum_forest /= "") then
    !   write (u, "(3x,A,A,A)") "MD5 sum (forest)      = ', &
    !   object%md5sum_forest, "'
    ! end if
    write (u, "(1x,A)") "Phase-space configuration parameters:"
    call phs_parameters_write (object%par, u)
    call object%mapping_defaults%write (u)
  end subroutine phs_wood_config_write

```

Print the PHS forest contents.

```

<PHS wood: phs wood config: TBP>+≡
  procedure :: write_forest => phs_wood_config_write_forest
<PHS wood: procedures>+≡
  subroutine phs_wood_config_write_forest (object, unit)
    class(phs_wood_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    call phs_forest_write (object%forest, u)
  end subroutine phs_wood_config_write_forest

```

Set the phase-space parameters that the configuration generator requests.

```

<PHS wood: phs wood config: TBP>+≡
  procedure :: set_parameters => phs_wood_config_set_parameters
<PHS wood: procedures>+≡
  subroutine phs_wood_config_set_parameters (phs_config, par)
    class(phs_wood_config_t), intent(inout) :: phs_config
    type(phs_parameters_t), intent(in) :: par
    phs_config%par = par
  end subroutine phs_wood_config_set_parameters

```

Enable the generation of channel equivalences (when calling `configure`).

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: enable_equivalences => phs_wood_config_enable_equivalences

(PHS wood: procedures)+≡
  subroutine phs_wood_config_enable_equivalences (phs_config)
    class(phs_wood_config_t), intent(inout) :: phs_config
    phs_config%use_equivalences = .true.
  end subroutine phs_wood_config_enable_equivalences

```

Set the phase-space mapping parameters that the configuration generator requests.

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: set_mapping_defaults => phs_wood_config_set_mapping_defaults

(PHS wood: procedures)+≡
  subroutine phs_wood_config_set_mapping_defaults (phs_config, mapping_defaults)
    class(phs_wood_config_t), intent(inout) :: phs_config
    type(mapping_defaults_t), intent(in) :: mapping_defaults
    phs_config%mapping_defaults = mapping_defaults
  end subroutine phs_wood_config_set_mapping_defaults

```

Define the input stream for the phase-space file as an open logical unit. The unit must be connected.

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: set_input => phs_wood_config_set_input

(PHS wood: procedures)+≡
  subroutine phs_wood_config_set_input (phs_config, unit)
    class(phs_wood_config_t), intent(inout) :: phs_config
    integer, intent(in) :: unit
    phs_config%io_unit = unit
    rewind (unit)
  end subroutine phs_wood_config_set_input

```

## 11.8.2 Phase-space generation

This subroutine generates a phase space configuration using the `cascades` module. Note that this may take time, and the `cascade_set` subobject may consume a large amount of memory.

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: generate_phase_space => phs_wood_config_generate_phase_space

(PHS wood: procedures)+≡
  subroutine phs_wood_config_generate_phase_space (phs_config)
    class(phs_wood_config_t), intent(inout) :: phs_config
    integer :: off_shell, extra_off_shell
    call msg_message ("Phase space: generating configuration ...")
    off_shell = phs_config%par%off_shell
    allocate (phs_config%cascade_set)
    do extra_off_shell = 0, max (phs_config%n_tot - 3, 0)
      phs_config%par%off_shell = off_shell + extra_off_shell
    end do
  end subroutine phs_wood_config_generate_phase_space

```

```

    call cascade_set_generate (phs_config%cascade_set, &
        phs_config%model, phs_config%n_in, phs_config%n_out, &
        phs_config%flv, &
        phs_config%par, fatal_beam_decay = .false.)
    if (cascade_set_is_valid (phs_config%cascade_set)) then
        exit
    else
        call msg_message ("Phase space: ... failed. &
            &Increasing phs_off_shell ...")
    end if
end do
if (cascade_set_is_valid (phs_config%cascade_set)) then
    call msg_message ("Phase space: ... success.")
else
    call msg_fatal ("Phase-space: generation failed")
end if
end subroutine phs_wood_config_generate_phase_space

```

Using the generated phase-space configuration, write an appropriate phase-space file to the stored (or explicitly specified) I/O unit.

*(PHS wood: phs wood config: TBP)*+≡

```

    procedure :: write_phase_space => phs_wood_config_write_phase_space

```

*(PHS wood: procedures)*+≡

```

subroutine phs_wood_config_write_phase_space (phs_config, unit)
    class(phs_wood_config_t), intent(in) :: phs_config
    integer, intent(in), optional :: unit
    integer :: u
    if (allocated (phs_config%cascade_set)) then
        if (present (unit)) then
            u = unit
        else
            u = phs_config%io_unit
        end if
        write (u, "(1x,A,A)") "process ", char (phs_config%id)
        write (u, "(A)")
        call cascade_set_write_process_bincode_format (phs_config%cascade_set, u)
        write (u, "(A)")
        write (u, "(3x,A,A,A32,A)") "md5sum_process    = ", &
            '', phs_config%md5sum_process, ''
        write (u, "(3x,A,A,A32,A)") "md5sum_model_par = ", &
            '', phs_config%md5sum_model_par, ''
        write (u, "(3x,A,A,A32,A)") "md5sum_phs_config = ", &
            '', phs_config%md5sum_phs_config, ''
        call phs_parameters_write (phs_config%par, u)
        call cascade_set_write_file_format (phs_config%cascade_set, u)
    else
        call msg_fatal ("Phase-space configuration: &
            &no phase space object generated")
    end if
end subroutine phs_wood_config_write_phase_space

```

Clear the phase-space configuration. This is useful since the object may become

really large.

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: clear_phase_space => phs_wood_config_clear_phase_space

(PHS wood: procedures)+≡
  subroutine phs_wood_config_clear_phase_space (phs_config)
    class(phs_wood_config_t), intent(inout) :: phs_config
    if (allocated (phs_config%cascade_set)) then
      call cascade_set_final (phs_config%cascade_set)
      deallocate (phs_config%cascade_set)
    end if
  end subroutine phs_wood_config_clear_phase_space

```

### 11.8.3 Phase-space configuration

We read the phase-space configuration from the stored I/O unit. If this is not set, we assume that we have to generate a phase space configuration. When done, we open a scratch file and write the configuration.

```

(PHS wood: phs wood config: TBP)+≡
  procedure :: configure => phs_wood_config_configure

(PHS wood: procedures)+≡
  subroutine phs_wood_config_configure &
    (phs_config, sqrts, sqrts_fixed, cm_frame, azimuthal_dependence, rebuild)
    class(phs_wood_config_t), intent(inout) :: phs_config
    real(default), intent(in) :: sqrts
    logical, intent(in), optional :: sqrts_fixed
    logical, intent(in), optional :: cm_frame
    logical, intent(in), optional :: azimuthal_dependence
    logical, intent(in), optional :: rebuild
    type(string_t) :: filename
    logical :: variable_limits
    logical :: ok, exist, found, match, rebuild_phs
    integer :: g, c0, c1, n
    phs_config%sqrts = sqrts
    phs_config%par%sqrts = sqrts
    if (present (sqrts_fixed)) &
      phs_config%sqrts_fixed = sqrts_fixed
    if (present (cm_frame)) &
      phs_config%cm_frame = cm_frame
    if (present (azimuthal_dependence)) &
      phs_config%azimuthal_dependence = azimuthal_dependence
    if (present (rebuild)) then
      rebuild_phs = rebuild
    else
      rebuild_phs = .true.
    end if
    phs_config%md5sum_forest = ""
    call phs_config%compute_md5sum ()
    if (phs_config%io_unit == 0) then
      filename = phs_config%id // ".phs"
      if (.not. rebuild_phs) then
        call phs_config%check_phs_file (exist, found, match)

```

```

        rebuild_phs = .not. (exist .and. found .and. match)
    end if
    if (rebuild_phs) then
        call phs_config%generate_phase_space ()
        phs_config%io_unit = free_unit ()
        if (phs_config%id /= "") then
            call msg_message ("Phase space: writing configuration file '" &
                // char (filename) // "'")
            open (phs_config%io_unit, file = char (filename), &
                status = "replace", action = "readwrite")
        else
            open (phs_config%io_unit, status = "scratch", action = "readwrite")
        end if
        call phs_config%write_phase_space ()
        rewind (phs_config%io_unit)
    else
        call msg_message ("Phase space: keeping configuration file '" &
            // char (filename) // "'")
    end if
end if
if (phs_config%io_unit == 0) then
    ok = .true.
else
    call phs_forest_read (phs_config%forest, phs_config%io_unit, &
        phs_config%id, phs_config%n_in, phs_config%n_out, &
        phs_config%model, ok)
    if (.not. phs_config%io_unit_keep_open) then
        close (phs_config%io_unit)
        phs_config%io_unit = 0
    end if
end if
if (ok) then
    call phs_forest_set_flavors (phs_config%forest, phs_config%flv(:,1))
    variable_limits = .not. phs_config%cm_frame
    call phs_forest_set_parameters &
        (phs_config%forest, phs_config%mapping_defaults, variable_limits)
    call phs_forest_setup_prt_combinations (phs_config%forest)
    phs_config%n_channel = phs_forest_get_n_channels (phs_config%forest)
    phs_config%n_par = phs_forest_get_n_parameters (phs_config%forest)
    allocate (phs_config%channel (phs_config%n_channel))
    if (phs_config%use_equivalences) then
        call phs_forest_set_equivalences (phs_config%forest)
        call phs_forest_get_equivalences (phs_config%forest, &
            phs_config%channel, phs_config%azimuthal_dependence)
        phs_config%provides_equivalences = .true.
    end if
    allocate (phs_config%chain (phs_config%n_channel), source = 0)
    do g = 1, phs_forest_get_n_groves (phs_config%forest)
        call phs_forest_get_grove_bounds (phs_config%forest, g, c0, c1, n)
        phs_config%chain (c0:c1) = g
    end do
    phs_config%provides_chains = .true.
    call phs_config%compute_md5sum_forest ()
else

```

```

        write (msg_buffer, "(A,A,A)") &
            "Phase space: process '", &
            char (phs_config%id), "' not found in configuration file"
        call msg_fatal ()
    end if
end subroutine phs_wood_config_configure

```

The MD5 sum of the forest is computed in addition to the MD5 sum of the configuration. The reason is that the forest may depend on a user-provided external file. On the other hand, this MD5 sum encodes all information that is relevant for further processing. Therefore, the `get_md5sum` method returns this result, once it is available.

```

<PHS wood: phs wood config: TBP>+≡
    procedure :: compute_md5sum_forest => phs_wood_config_compute_md5sum_forest

<PHS wood: procedures>+≡
    subroutine phs_wood_config_compute_md5sum_forest (phs_config)
        class(phs_wood_config_t), intent(inout) :: phs_config
        integer :: u
        u = free_unit ()
        open (u, status = "scratch", action = "readwrite")
        call phs_config%write_forest (u)
        rewind (u)
        phs_config%md5sum_forest = md5sum (u)
        close (u)
    end subroutine phs_wood_config_compute_md5sum_forest

```

Return the most relevant MD5 sum. This overrides the method of the base type.

```

<PHS wood: phs wood config: TBP>+≡
    procedure :: get_md5sum => phs_wood_config_get_md5sum

<PHS wood: procedures>+≡
    function phs_wood_config_get_md5sum (phs_config) result (md5sum)
        class(phs_wood_config_t), intent(in) :: phs_config
        character(32) :: md5sum
        if (phs_config%md5sum_forest /= "") then
            md5sum = phs_config%md5sum_forest
        else
            md5sum = phs_config%md5sum_phs_config
        end if
    end function phs_wood_config_get_md5sum

```

Check whether a phase-space configuration for the current process exists. We look for the phase-space file that should correspond to the current process. If we find it, we check the MD5 sums stored in the file against the MD5 sums in the current configuration.

```

<PHS wood: phs wood config: TBP>+≡
    procedure :: check_phs_file => phs_wood_check_phs_file

<PHS wood: procedures>+≡
    subroutine phs_wood_check_phs_file (phs_config, exist, found, match)
        class(phs_wood_config_t), intent(inout) :: phs_config

```

```

logical, intent(out) :: exist
logical, intent(out) :: found
logical, intent(out) :: match
type(string_t) :: filename
integer :: u
filename = phs_config%id // ".phs"
inquire (file = char (filename), exist = exist)
if (exist) then
  u = free_unit ()
  open (u, file = char (filename), action = "read", status = "old")
  call phs_forest_read (phs_config%forest, u, &
    phs_config%id, phs_config%n_in, phs_config%n_out, &
    phs_config%model, found, &
    phs_config%md5sum_process, &
    phs_config%md5sum_model_par, &
    phs_config%md5sum_phs_config, &
    match = match)
  close (u)
else
  found = .false.
  match = .false.
end if
end subroutine phs_wood_check_phs_file

```

Startup message, after configuration is complete.

```

<PHS wood: phs wood config: TBP>+≡
  procedure :: startup_message => phs_wood_config_startup_message

<PHS wood: procedures>+≡
  subroutine phs_wood_config_startup_message (phs_config, unit)
    class(phs_wood_config_t), intent(in) :: phs_config
    integer, intent(in), optional :: unit
    call phs_config%base_startup_message (unit)
    write (msg_buffer, "(A,2(1x,I0,1x,A))") &
      "Phase space: wood"
    call msg_message (unit = unit)
  end subroutine phs_wood_config_startup_message

```

Match channels: currently we support only single-channel structure function setups.

(Note: we should extend the implementation to account for resonances in the s-channel.)

```

<PHS wood: phs wood config: TBP>+≡
  procedure :: match_channels => phs_wood_config_match_channels

<PHS wood: procedures>+≡
  subroutine phs_wood_config_match_channels (phs_config, sf, sf_channel)
    class(phs_wood_config_t), intent(inout) :: phs_config
    type(sf_config_t), dimension(:), intent(in), allocatable :: sf
    type(sf_channel_t), dimension(:), intent(in), allocatable :: sf_channel
    if (allocated (sf_channel)) then
      select case (size (sf_channel))
      case (1)
        phs_config%channel%sf_channel = 1

```

```

        case default
            call msg_fatal ("Phase-space configuration: &
                &multi-channel structure functions not supported yet.")
        end select
    end if
end subroutine phs_wood_config_match_channels

```

Allocate an instance: the actual phase-space object.

```

<PHS wood: phs wood config: TBP>+≡
    procedure, nopass :: allocate_instance => phs_wood_config_allocate_instance

<PHS wood: procedures>+≡
    subroutine phs_wood_config_allocate_instance (phs)
        class(phs_t), intent(inout), pointer :: phs
        allocate (phs_wood_t :: phs)
    end subroutine phs_wood_config_allocate_instance

```

#### 11.8.4 Kinematics implementation

We generate  $\cos\theta$  and  $\phi$  uniformly, covering the solid angle.

```

<PHS wood: public>+≡
    public :: phs_wood_t

<PHS wood: types>+≡
    type, extends (phs_t) :: phs_wood_t
        real(default) :: sqrts = 0
        type(phs_forest_t) :: forest
    contains
        <PHS wood: phs wood: TBP>
    end type phs_wood_t

```

Output. The `verbose` setting is irrelevant, we just display the contents of the base object.

```

<PHS wood: phs wood: TBP>≡
    procedure :: write => phs_wood_write

<PHS wood: procedures>+≡
    subroutine phs_wood_write (object, unit, verbose)
        class(phs_wood_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer :: u
        logical :: verb
        u = output_unit (unit)
        verb = .false.; if (present (verbose)) verb = verbose
        call object%base_write (u)
    end subroutine phs_wood_write

```

Write the forest separately.

```

<PHS wood: phs wood: TBP>+≡
    procedure :: write_forest => phs_wood_write_forest

```



```

<PHS wood: procedures>+≡
  subroutine phs_wood_write_forest (object, unit)
    class(phs_wood_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    call phs_forest_write (object%forest, u)
  end subroutine phs_wood_write_forest

```

Finalizer.

```

<PHS wood: phs wood: TBP>+≡
  procedure :: final => phs_wood_final
<PHS wood: procedures>+≡
  subroutine phs_wood_final (object)
    class(phs_wood_t), intent(inout) :: object
    call phs_forest_final (object%forest)
  end subroutine phs_wood_final

```

Initialization. We allocate arrays (`base_init`) and adjust the phase-space volume. The two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \quad (11.61)$$

independent of the particle masses.

```

<PHS wood: phs wood: TBP>+≡
  procedure :: init => phs_wood_init
<PHS wood: procedures>+≡
  subroutine phs_wood_init (phs, phs_config)
    class(phs_wood_t), intent(out) :: phs
    class(phs_config_t), intent(in), target :: phs_config
    call phs%base_init (phs_config)
    select type (phs_config)
      type is (phs_wood_config_t)
        phs%forest = phs_config%forest
    end select
  end subroutine phs_wood_init

```

### 11.8.5 Evaluation

We compute the outgoing momenta from the incoming momenta and the input parameter set `r_in` in channel `r_in`. We also compute the `r` parameters and Jacobians `f` for all other channels.

We do *not* need to apply a transformation from/to the c.m. frame, because in `phs_base` the momenta are already boosted to the c.m. frame before assigning them in the `phs` object, and inversely boosted when extracting them.

Note: We should postpone computation of the complete `r` and `f` arrays until after cut evaluation. This was done in the previous WHIZARD version, to save computing time for events which fail cut evaluation.

```

<PHS wood: phs wood: TBP>+≡
  procedure :: evaluate => phs_wood_evaluate

```

```

(PHS wood: procedures)+≡
subroutine phs_wood_evaluate (phs, c_in, r_in)
  class(phs_wood_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  logical :: ok
  if (phs%p_defined) then
    call phs_forest_set_prt_in (phs%forest, phs%p)
    phs%r(:,c_in) = r_in
    call phs_forest_evaluate_phase_space (phs%forest, &
      c_in, phs%active_channel, &
      phs%sqrts_hat, phs%r, phs%f, phs%volume, phs%r_defined)
    if (phs%r_defined) then
      phs%q = phs_forest_get_momenta_out (phs%forest)
    end if
    phs%q_defined = phs%r_defined
  end if
end subroutine phs_wood_evaluate

```

Inverse evaluation.

```

(PHS wood: phs wood: TBP)+≡
procedure :: inverse => phs_wood_inverse

(PHS wood: procedures)+≡
subroutine phs_wood_inverse (phs)
  class(phs_wood_t), intent(inout) :: phs
  integer :: n_channel
  real(default), dimension(:), allocatable :: x
  if (phs%p_defined .and. phs%q_defined) then
    call phs_forest_set_prt_in (phs%forest, phs%p)
    call phs_forest_set_prt_out (phs%forest, phs%q)
    call phs_forest_recover_channel (phs%forest, &
      1, &
      phs%sqrts_hat, phs%r, phs%f, phs%volume)
    call phs_forest_evaluate_other_channels (phs%forest, &
      1, phs%active_channel, &
      phs%sqrts_hat, phs%r, phs%f)
    phs%r_defined = .true.
  end if
end subroutine phs_wood_inverse

```

### 11.8.6 Unit tests

```

(PHS wood: public)+≡
public :: phs_wood_test

(PHS wood: tests)≡
subroutine phs_wood_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
(PHS wood: execute tests)
end subroutine phs_wood_test

```

## Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the azimuthal\_dependence flag.

This auxiliary routine writes a phase-space configuration file to unit `u_phs`.

*(PHS wood: public)*+≡

```
public :: write_test_phs_file
```

*(PHS wood: tests)*+≡

```
subroutine write_test_phs_file (u_phs, procname)
  integer, intent(in) :: u_phs
  type(string_t), intent(in), optional :: procname
  if (present (procname)) then
    write (u_phs, "(A,A)") "process ", char (procname)
  else
    write (u_phs, "(A)") "process testproc"
  end if
  write (u_phs, "(A,A)") "  md5sum_process   = ", ''''
  write (u_phs, "(A,A)") "  md5sum_model_par = ", ''''
  write (u_phs, "(A,A)") "  md5sum_phs_config = ", ''''
  write (u_phs, "(A)") "  sqrts           = 1000"
  write (u_phs, "(A)") "  m_threshold_s = 50"
  write (u_phs, "(A)") "  m_threshold_t = 100"
  write (u_phs, "(A)") "  off_shell = 2"
  write (u_phs, "(A)") "  t_channel = 6"
  write (u_phs, "(A)") "  keep_nonresonant = T"
  write (u_phs, "(A)") "  grove #1"
  write (u_phs, "(A)") "  tree 3"
end subroutine write_test_phs_file
```

*(PHS wood: execute tests)*≡

```
call test (phs_wood_1, "phs_wood_1", &
  "phase-space configuration", &
  u, results)
```

*(PHS wood: tests)*+≡

```
subroutine phs_wood_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_t), pointer :: model
  type(process_constants_t) :: process_data
  class(phs_config_t), allocatable :: phs_data
  type(mapping_defaults_t) :: mapping_defaults
  real(default) :: sqrts
  integer :: u_phs, iostat
  character(32) :: buffer

  write (u, "(A)") "* Test output: phs_wood_1"
  write (u, "(A)") "* Purpose: initialize and display &
    &phase-space configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("Test"), &
```

```

var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_1"), process_data)

write (u, "(A)")  "* Create a scratch phase-space file"
write (u, "(A)")

u_phs = free_unit ()
open (u_phs, status = "scratch", action = "readwrite")
call write_test_phs_file (u_phs, var_str ("phs_wood_1"))
rewind (u_phs)
do
  read (u_phs, "(A)", iostat = iostat)  buffer
  if (iostat /= 0)  exit
  write (u, "(A)") trim (buffer)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup phase-space configuration object"
write (u, "(A)")

mapping_defaults%step_mapping = .false.

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)
select type (phs_data)
type is (phs_wood_config_t)
  call phs_data%set_input (u_phs)
  call phs_data%set_mapping_defaults (mapping_defaults)
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
  call phs_data%write_forest (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

close (u_phs)
call phs_data%final ()
call model_list_final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_1"

end subroutine phs_wood_1

```

## Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

```

<PHS wood: execute tests>+≡
  call test (phs_wood_2, "phs_wood_2", &
    "phase-space evaluation", &
    u, results)

<PHS wood: tests>+≡
  subroutine phs_wood_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(process_constants_t) :: process_data
    real(default) :: sqrts, E
    class(phs_config_t), allocatable, target :: phs_data
    class(phs_t), pointer :: phs => null ()
    type(vector4_t), dimension(2) :: p, q
    integer :: u_phs

    write (u, "(A)")  "* Test output: phs_wood_2"
    write (u, "(A)")  "* Purpose: test simple single-channel phase space"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)
    call flavor_init (flv, 25, model)

    write (u, "(A)")  "* Initialize a process and a matching &
      &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_wood_2"), process_data)
    u_phs = free_unit ()
    open (u_phs, status = "scratch", action = "readwrite")
    call write_test_phs_file (u_phs, var_str ("phs_wood_2"))
    rewind (u_phs)

    allocate (phs_wood_config_t :: phs_data)
    call phs_data%init (process_data)
    select type (phs_data)
    type is (phs_wood_config_t)
      call phs_data%set_input (u_phs)
    end select

```

```

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
&for x = 0.125, 0.5"
write (u, "(A)")

call phs%evaluate (1, [0.125_default, 0.5_default])
call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_t)
    call phs%write_forest (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call phs%final ()
deallocate (phs)

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

```

```

call phs%inverse ()
call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_t)
    call phs%write_forest (u)
end select

call phs%final ()
deallocate (phs)

close (u_phs)
call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_2"

end subroutine phs_wood_2

```

## Phase-space generation

Generate phase space for a simple process.

```

<PHS wood: execute tests>+≡
    call test (phs_wood_3, "phs_wood_3", &
        "phase-space generation", &
        u, results)

<PHS wood: tests>+≡
subroutine phs_wood_3 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    type(phs_parameters_t) :: phs_par
    class(phs_config_t), allocatable :: phs_data
    integer :: iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: phs_wood_3"
    write (u, "(A)")  "* Purpose: generate a phase-space configuration"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("Test"), &
        var_str ("Test.mdl"), os_data, model)

    call syntax_phs_forest_init ()

    write (u, "(A)")  "* Initialize a process and phase-space parameters"
    write (u, "(A)")

```

```

call init_test_process_data (var_str ("phs_wood_3"), process_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    phs_data%io_unit_keep_open = .true.
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a scratch phase-space file"
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)

select type (phs_data)
type is (phs_wood_config_t)
    rewind (phs_data%io_unit)
    do
        read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
        if (iostat /= 0) exit
        write (u, "(A)") trim (buffer)
    end do
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_3"

end subroutine phs_wood_3

```

## Nontrivial process

Generate phase space for a  $2 \rightarrow 3$  process.

```

<PHS wood: execute tests>+≡
    call test (phs_wood_4, "phs_wood_4", &
        "nontrivial process", &
        u, results)

<PHS wood: tests>+≡
    subroutine phs_wood_4 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_t), pointer :: model
        type(process_constants_t) :: process_data
        type(phs_parameters_t) :: phs_par

```



```

class(phs_config_t), allocatable, target :: phs_data
integer :: iostat
character(80) :: buffer
class(phs_t), pointer :: phs => null ()
real(default) :: E, pL
type(vector4_t), dimension(2) :: p
type(vector4_t), dimension(3) :: q

write (u, "(A)")  "* Test output: phs_wood_4"
write (u, "(A)")  "* Purpose: generate a phase-space configuration"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process and phase-space parameters"
write (u, "(A)")

process_data%id = "phs_wood_4"
process_data%model_name = "Test"
process_data%n_in = 2
process_data%n_out = 3
process_data%n_flv = 1
allocate (process_data%flv_state (process_data%n_in + process_data%n_out, &
    process_data%n_flv))
process_data%flv_state(:,1) = [25, 25, 25, 6, -6]

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    phs_data%io_unit_keep_open = .true.
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a scratch phase-space file"
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)

select type (phs_data)
type is (phs_wood_config_t)
    rewind (phs_data%io_unit)
    do
        read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
        if (iostat /= 0)  exit
        write (u, "(A)") trim (buffer)
    end do

```

```

        end do
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Initialize the phase-space instance"
    write (u, "(A)")

    call phs_data%allocate_instance (phs)
    call phs%init (phs_data)

    write (u, "(A)")  "* Set incoming momenta"
    write (u, "(A)")

    select type (phs_data)
    type is (phs_wood_config_t)
        E = phs_data%sqrts / 2
        pL = sqrt (E**2 - flavor_get_mass (phs_data%flv(1,1))**2)
    end select
    p(1) = vector4_moving (E, pL, 3)
    p(2) = vector4_moving (E, -pL, 3)

    call phs%set_incoming_momenta (p)
    call phs%compute_flux ()

    write (u, "(A)")  "* Compute phase-space point &
        &for x = 0.1, 0.2, 0.3, 0.4, 0.5"
    write (u, "(A)")

    call phs%evaluate (1, &
        [0.1_default, 0.2_default, 0.3_default, 0.4_default, 0.5_default])
    call phs%write (u)
!     write (u, "(A)")
!     select type (phs)
!     type is (phs_wood_t)
!         call phs%write_forest (u)
!     end select

    write (u, "(A)")
    write (u, "(A)")  "* Inverse kinematics"
    write (u, "(A)")

    call phs%get_outgoing_momenta (q)
    call phs%final ()
    deallocate (phs)

    call phs_data%allocate_instance (phs)
    call phs%init (phs_data)

    call phs%set_incoming_momenta (p)
    call phs%compute_flux ()
    call phs%set_outgoing_momenta (q)

    call phs%inverse ()
    call phs%write (u)

```

```

!      write (u, "(A)")
!      select type (phs)
!      type is (phs_wood_t)
!          call phs%write_forest (u)
!      end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call phs%final ()
deallocate (phs)

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_4"

end subroutine phs_wood_4

```

## Equivalences

Generate phase space for a simple process, including channel equivalences.

```

<PHS wood: execute tests>+≡
    call test (phs_wood_5, "phs_wood_5", &
               "equivalences", &
               u, results)

<PHS wood: tests>+≡
    subroutine phs_wood_5 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_t), pointer :: model
        type(process_constants_t) :: process_data
        type(phs_parameters_t) :: phs_par
        class(phs_config_t), allocatable :: phs_data
        integer :: iostat
        character(80) :: buffer

        write (u, "(A)")  "* Test output: phs_wood_5"
        write (u, "(A)")  "* Purpose: generate a phase-space configuration"
        write (u, "(A)")

        call os_data_init (os_data)
        call syntax_model_file_init ()
        call model_list_read_model (var_str ("Test"), &
                                   var_str ("Test.mdl"), os_data, model)

        call syntax_phs_forest_init ()

        write (u, "(A)")  "* Initialize a process and phase-space parameters"
        write (u, "(A)")
    end subroutine phs_wood_5

```

```

call init_test_process_data (var_str ("phs_wood_5"), process_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    call phs_data%enable_equivalences ()
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a scratch phase-space file"
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)
call phs_data%write (u)
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%write_forest (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_5"

end subroutine phs_wood_5

```

## MD5 sum checks

Generate phase space for a simple process. Repeat this with and without parameter change.

```

<PHS wood: execute tests>+≡
call test (phs_wood_6, "phs_wood_6", &
    "phase-space generation", &
    u, results)

<PHS wood: tests>+≡
subroutine phs_wood_6 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    type(phs_parameters_t) :: phs_par
    class(phs_config_t), allocatable :: phs_data
    logical :: exist, found, match

```

```

integer :: u_phs
character(*), parameter :: filename = "phs_wood_6_p.phs"
type(var_list_t), pointer :: var_list

write (u, "(A)")  "* Test output: phs_wood_6"
write (u, "(A)")  "* Purpose: generate and check phase-space file"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process and phase-space parameters"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_6"), process_data)
process_data%id = "phs_wood_6_p"
process_data%md5sum = "1234567890abcdef1234567890abcdef"
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
end select

write (u, "(A)")  "* Remove previous phs file, if any"
write (u, "(A)")

inquire (file = filename, exist = exist)
if (exist) then
    u_phs = free_unit ()
    open (u_phs, file = filename, action = "write")
    close (u_phs, status = "delete")
end if

write (u, "(A)")  "* Check phase-space file (should fail)"
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%check_phs_file (exist, found, match)
    write (u, "(1x,A,L1)") "exist = ", exist
    write (u, "(1x,A,L1)") "found = ", found
    write (u, "(1x,A,L1)") "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a phase-space file"
write (u, "(A)")

```

```

call phs_data%configure (phs_par%sqrts)

write (u, "(1x,A,A,A)") "MD5 sum (process)    = '", &
    phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)") "MD5 sum (model par)  = '", &
    phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)") "MD5 sum (phs config) = '", &
    phs_data%md5sum_phs_config, "'"

write (u, "(A)")
write (u, "(A)")  "* Check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)
phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    phs_data%sqrts = phs_par%sqrts
    phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)") "MD5 sum (process)    = '", &
    phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)") "MD5 sum (model par)  = '", &
    phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)") "MD5 sum (phs config) = '", &
    phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%check_phs_file (exist, found, match)
    write (u, "(1x,A,L1)") "exist = ", exist
    write (u, "(1x,A,L1)") "found = ", found
    write (u, "(1x,A,L1)") "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify sqrts and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data)
phs_par%sqrts = 500
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)

```



```

write (u, "(A)")  "* Modify phs parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
process_data%md5sum = "1234567890abcdef1234567890abcdef"
call phs_data%init (process_data)
phs_par%sqrts = 1000
phs_par%off_shell = 17
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    phs_data%sqrts = phs_par%sqrts
    phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
    phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
    phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
    phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%check_phs_file (exist, found, match)
    write (u, "(1x,A,L1)")  "exist = ", exist
    write (u, "(1x,A,L1)")  "found = ", found
    write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify model parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, var_str ("ms"), 100._default, &
    is_known = .true.)
call phs_data%init (process_data)
phs_par%sqrts = 1000
phs_par%off_shell = 1
select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%set_parameters (phs_par)
    phs_data%sqrts = phs_par%sqrts
    phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

```



```

write (u, "(1x,A,A,A)") "MD5 sum (process)    = ', &
    phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)") "MD5 sum (model par) = ', &
    phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)") "MD5 sum (phs config) = ', &
    phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
    call phs_data%check_phs_file (exist, found, match)
    write (u, "(1x,A,L1)") "exist = ", exist
    write (u, "(1x,A,L1)") "found = ", found
    write (u, "(1x,A,L1)") "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call phs_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_6"

end subroutine phs_wood_6

```

## Chapter 12

# Random Number Generation

The integration and event generation modules need a random-number generator. Here, we provide an abstract interface for a random-number generator and a concrete implementation.

**Module rng\_base:** Abstract type for interfacing or implementing a r.n.g.

**Module rng\_tao:** Implementation as an interface to the TAO random number generator which the VAMP package provides.

### 12.1 Generic Random-Number Generator

```
<rng_base.f90>≡  
<File header>  
  
module rng_base  
  
    use kinds !NODEP!  
<Use file utils>  
    use unit_tests  
  
<Standard module head>  
  
<RNG base: public>  
  
<RNG base: types>  
  
<RNG base: interfaces>  
  
<RNG base: test types>  
  
contains  
  
<RNG base: procedures>  
  
<RNG base: tests>
```

```
end module rng_base
```

### 12.1.1 Generator type

The `rng` object is actually the state of the random-number generator. The methods `initialize/reset` and `call` the generator for this state.

```
<RNG base: public>≡
  public :: rng_t

<RNG base: types>≡
  type, abstract :: rng_t
  contains
    <RNG base: rng: TBP>
  end type rng_t
```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The behavior without seed is not defined, however.

```
<RNG base: rng: TBP>≡
  procedure (rng_init), deferred :: init

<RNG base: interfaces>≡
  abstract interface
    subroutine rng_init (rng, seed)
      import
        class(rng_t), intent(out) :: rng
        integer, intent(in), optional :: seed
    end subroutine rng_init
  end interface
```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

```
<RNG base: rng: TBP>+≡
  procedure (rng_final), deferred :: final

<RNG base: interfaces>+≡
  abstract interface
    subroutine rng_final (rng)
      import
        class(rng_t), intent(inout) :: rng
    end subroutine rng_final
  end interface
```

Output. We should, at least, identify the generator.

```
<RNG base: rng: TBP>+≡
  procedure (rng_write), deferred :: write

<RNG base: interfaces>+≡
  abstract interface
    subroutine rng_write (rng, unit)
      import
        class(rng_t), intent(in) :: rng
```

```

        integer, intent(in), optional :: unit
    end subroutine rng_write
end interface

```

These routines generate a single and an array of default-precision random numbers, respectively.

```

<RNG base: rng: TBP>+≡
    generic :: generate => generate_single, generate_array
    procedure (rng_generate_single), deferred :: generate_single
    procedure (rng_generate_array), deferred :: generate_array

<RNG base: interfaces>+≡
    abstract interface
        subroutine rng_generate_single (rng, x)
            import
            class(rng_t), intent(inout) :: rng
            real(default), intent(out) :: x
        end subroutine rng_generate_single
    end interface

    abstract interface
        subroutine rng_generate_array (rng, x)
            import
            class(rng_t), intent(inout) :: rng
            real(default), dimension(:), intent(out) :: x
        end subroutine rng_generate_array
    end interface

```

### 12.1.2 Unit tests

```

<RNG base: public>+≡
    public :: rng_base_test

<RNG base: tests>≡
    subroutine rng_base_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <RNG base: execute tests>
    end subroutine rng_base_test

```

#### Test generator

The test 'mock' random generator generates a repeating series with the numbers 0.1, 0.3, 0.5, 0.7, 0.9. It has an integer stored as state. The integer must be one of 1, 3, 5, 7, 9.

```

<RNG base: public>+≡
    public :: rng_test_t

```

```

(RNG base: test types)≡
  type, extends (rng_t) :: rng_test_t
    integer :: state = 1
  contains
    procedure :: write => rng_test_write
    procedure :: init => rng_test_init
    procedure :: final => rng_test_final
    procedure :: generate_single => rng_test_generate_single
    procedure :: generate_array => rng_test_generate_array
  end type rng_test_t

```

Output. The state is a single number, so print it.

```

(RNG base: tests)+≡
  subroutine rng_test_write (rng, unit)
    class(rng_test_t), intent(in) :: rng
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A,I0,A)") "Random-number generator: &
      &test (state = ", rng%state, ")"
  end subroutine rng_test_write

```

The default seed is 1.

```

(RNG base: tests)+≡
  subroutine rng_test_init (rng, seed)
    class(rng_test_t), intent(out) :: rng
    integer, intent(in), optional :: seed
    if (present (seed)) rng%state = seed
  end subroutine rng_test_init

```

Nothing to finalize:

```

(RNG base: tests)+≡
  subroutine rng_test_final (rng)
    class(rng_test_t), intent(inout) :: rng
  end subroutine rng_test_final

```

Generate a single number and advance the state.

```

(RNG base: tests)+≡
  subroutine rng_test_generate_single (rng, x)
    class(rng_test_t), intent(inout) :: rng
    real(default), intent(out) :: x
    x = rng%state / 10._default
    rng%state = mod (rng%state + 2, 10)
  end subroutine rng_test_generate_single

```

The array generator calls the single-item generator multiple times.

```

(RNG base: tests)+≡
  subroutine rng_test_generate_array (rng, x)
    class(rng_test_t), intent(inout) :: rng
    real(default), dimension(:), intent(out) :: x
    integer :: i

```

```

do i = 1, size (x)
  call rng%generate (x(i))
end do
end subroutine rng_test_generate_array

```

## Generator check

Initialize the generator and draw random numbers.

```

(RNG base: execute tests)≡
  call test (rng_base_1, "rng_base_1", &
    "rng initialization and call", &
    u, results)

(RNG base: tests)+≡
  subroutine rng_base_1 (u)
    integer, intent(in) :: u
    class(rng_t), allocatable, target :: rng

    real(default) :: x
    real(default), dimension(2) :: x2

    write (u, "(A)")  "* Test output: rng_base_1"
    write (u, "(A)")  "* Purpose: initialize and call a test random-number &
      &generator"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize generator"
    write (u, "(A)")

    allocate (rng_test_t :: rng)
    call rng%init (3)

    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Get random number"
    write (u, "(A)")

    call rng%generate (x)
    write (u, "(A,2(1x,F9.7))")  "x =", x

    write (u, "(A)")
    write (u, "(A)")  "* Get random number pair"
    write (u, "(A)")

    call rng%generate (x2)
    write (u, "(A,2(1x,F9.7))")  "x =", x2

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: rng_base_1"

end subroutine rng_base_1

```

## 12.2 TAO Random-Number Generator

This module provides an implementation for the generic random-number generator. Actually, we interface the TAO random-number generator which is available via the VAMP package.

```

⟨rng_tao.f90⟩≡
  ⟨File header⟩

  module rng_tao

    use kinds !NODEP!
    ⟨Use file utils⟩
    use unit_tests
    use tao_random_numbers !NODEP!

    use rng_base

    ⟨Standard module head⟩

    ⟨RNG tao: public⟩

    ⟨RNG tao: types⟩

    contains

    ⟨RNG tao: procedures⟩

    ⟨RNG tao: tests⟩

    end module rng_tao

```

### 12.2.1 Generator type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

We keep the seed, in case we want to recover it later, and count the number of calls since seeding.

```

⟨RNG tao: public⟩≡
  public :: rng_tao_t

⟨RNG tao: types⟩≡
  type, extends (rng_t) :: rng_tao_t
    integer :: seed = 0
    integer :: n_calls = 0
    type(tao_random_state) :: state
  contains

```

```

    <RNG tao: rng tao: TBP>
end type rng_tao_t

```

Output: Display seed and number of calls.

```

<RNG tao: rng tao: TBP>≡
    procedure :: write => rng_tao_write

<RNG tao: procedures>≡
    subroutine rng_tao_write (rng, unit)
        class(rng_tao_t), intent(in) :: rng
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "TAO random-number generator:"
        write (u, "(3x,A,I0)") "seed = ", rng%seed
        write (u, "(3x,A,I0)") "calls = ", rng%n_calls
    end subroutine rng_tao_write

```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The default seed is 0.

```

<RNG tao: rng tao: TBP>+≡
    procedure :: init => rng_tao_init

<RNG tao: procedures>+≡
    subroutine rng_tao_init (rng, seed)
        class(rng_tao_t), intent(out) :: rng
        integer, intent(in), optional :: seed
        if (present (seed)) rng%seed = seed
        call tao_random_create (rng%state, rng%seed)
    end subroutine rng_tao_init

```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

```

<RNG tao: rng tao: TBP>+≡
    procedure :: final => rng_tao_final

<RNG tao: procedures>+≡
    subroutine rng_tao_final (rng)
        class(rng_tao_t), intent(inout) :: rng
        call tao_random_destroy (rng%state)
    end subroutine rng_tao_final

```

These routines generate a single and an array of default-precision random numbers, respectively.

We have to convert from explicit double to abstract default precision. Under normal conditions, both are equivalent, however. Unless, someone decides to do single precision, there is always an interface for `tao_random_numbers`.

```

<RNG tao: rng tao: TBP>+≡
    procedure :: generate_single => rng_tao_generate_single
    procedure :: generate_array => rng_tao_generate_array

```



```

<RNG tao: procedures>+=
  subroutine rng_tao_generate_single (rng, x)
    class(rng_tao_t), intent(inout) :: rng
    real(default), intent(out) :: x
    real(default) :: r
    call tao_random_number (rng%state, r)
    x = r
    rng%n_calls = rng%n_calls + 1
  end subroutine rng_tao_generate_single

  subroutine rng_tao_generate_array (rng, x)
    class(rng_tao_t), intent(inout) :: rng
    real(default), dimension(:), intent(out) :: x
    real(default) :: r
    integer :: i
    do i = 1, size (x)
      call tao_random_number (rng%state, r)
      x(i) = r
    end do
    rng%n_calls = rng%n_calls + size (x)
  end subroutine rng_tao_generate_array

```

### 12.2.2 Unit tests

```

<RNG tao: public>+=
  public :: rng_tao_test

<RNG tao: tests>=
  subroutine rng_tao_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <RNG tao: execute tests>
  end subroutine rng_tao_test

```

#### Generator check

Initialize the generator and draw random numbers.

```

<RNG tao: execute tests>=
  call test (rng_tao_1, "rng_tao_1", &
    "rng initialization and call", &
    u, results)

<RNG tao: tests>+=
  subroutine rng_tao_1 (u)
    integer, intent(in) :: u
    class(rng_t), allocatable, target :: rng

    real(default) :: x
    real(default), dimension(2) :: x2

    write (u, "(A)")  "* Test output: rng_tao_1"
    write (u, "(A)")  "* Purpose: initialize and call the TAO random-number &

```

```

        &generator"
write (u, "(A)")

write (u, "(A)")  "* Initialize generator (default seed)"
write (u, "(A)")

allocate (rng_tao_t :: rng)
call rng%init ()

call rng%write (u)

write (u, "(A)")
write (u, "(A)")  "* Get random number"
write (u, "(A)")

call rng%generate (x)
write (u, "(A,2(1x,F9.7))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Get random number pair"
write (u, "(A)")

call rng%generate (x2)
write (u, "(A,2(1x,F9.7))")  "x =", x2

write (u, "(A)")
call rng%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call rng%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rng_tao_1"

end subroutine rng_tao_1

```

## Chapter 13

# Multi-Channel Integration

In this chapter we provide a generic module for handling multi-channel phase space and concrete implementations.

**Module mci\_base:** The abstract types and their methods. It provides a test integrator that is referenced in later unit tests.

**Module mci\_midpoint:** A simple integrator that uses the midpoint rule to sample the integrand uniformly over the unit hypercube. There is only one integration channel, so this can be matched only to single-channel phase space.

**Module mci\_vamp:** Interface for the VAMP package. (not implemented yet)

### 13.1 Generic Integrator

This module provides a multi-channel integrator (MCI) base type, a corresponding configuration type, and methods for integration and event generation.

```
<mci_base.f90>≡  
<File header>  
  
module mci_base  
  
    use kinds !NODEP!  
<Use strings>  
<Use file utils>  
    use diagnostics !NODEP!  
    use unit_tests  
  
    use phs_base  
    use rng_base  
  
<Standard module head>  
  
<MCI base: public>  
  
<MCI base: types>
```

*⟨MCI base: interfaces⟩*

*⟨MCI base: test types⟩*

**contains**

*⟨MCI base: procedures⟩*

*⟨MCI base: tests⟩*

**end module mci\_base**

### 13.1.1 MCI: integrator

The MCI object contains the methods for integration and event generation. For the actual work and data storage, it spawns an MCI instance object.

The base object contains the number of integration dimensions and the number of channels as configuration data. Further configuration data are stored in the concrete extensions.

The MCI sum contains all relevant information about the integrand. It can be used for comparing the current configuration against a previous one. If they match, we can skip an actual integration. (Implemented only for the VAMP version.)

There is a random-number generator (its state with associated methods) available as **rng**. It may or may not be used for integration. It will be used for event generation.

*⟨MCI base: public⟩*≡

**public :: mci\_t**

*⟨MCI base: types⟩*≡

```
type, abstract :: mci_t
  integer :: n_dim = 0
  integer :: n_channel = 0
  integer :: n_chain = 0
  integer, dimension(:), allocatable :: chain
  character(32) :: md5sum = ""
  logical :: integral_known = .false.
  logical :: error_known = .false.
  logical :: efficiency_known = .false.
  real(default) :: integral = 0
  real(default) :: error = 0
  real(default) :: efficiency = 0
  class(rng_t), allocatable :: rng
contains
  ⟨MCI base: mci: TBP⟩
end type mci_t
```

Finalizer: the random-number generator may need one.

*⟨MCI base: mci: TBP⟩*≡

```
procedure :: base_final => mci_final
procedure (mci_final), deferred :: final
```

```

<MCI base: procedures>+=
  subroutine mci_final (object)
    class(mci_t), intent(inout) :: object
    if (allocated (object%rng)) call object%rng%final ()
  end subroutine mci_final

```

Output: basic and extended output.

```

<MCI base: mci: TBP>+=
  procedure :: base_write => mci_write
  procedure (mci_write), deferred :: write

<MCI base: procedures>+=
  subroutine mci_write (object, unit)
    class(mci_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i, j
    u = output_unit (unit)
    if (object%integral_known) then
      write (u, "(3x,A,ES17.10)") "Integral" = ", object%integral
    end if
    if (object%error_known) then
      write (u, "(3x,A,ES17.10)") "Error" = ", object%error
    end if
    if (object%efficiency_known) then
      write (u, "(3x,A,ES17.10)") "Efficiency" = ", object%efficiency
    end if
    write (u, "(3x,A,I0)") "Number of channels" = ", object%n_channel
    write (u, "(3x,A,I0)") "Number of dimensions" = ", object%n_dim
    if (object%n_chain > 0) then
      write (u, "(3x,A,I0)") "Number of chains" = ", object%n_chain
      write (u, "(3x,A)") "Chains:"
      do i = 1, object%n_chain
        write (u, "(5x,I0,':')", advance = "no") i
        do j = 1, object%n_channel
          if (object%chain(j) == i) &
            write (u, "(1x,I0)", advance = "no") j
          end do
        write (u, "(A)")
      end do
    end if
  end subroutine mci_write

```

Print an informative message when starting integration.

```

<MCI base: mci: TBP>+=
  procedure (mci_startup_message), deferred :: startup_message
  procedure :: base_startup_message => mci_startup_message

<MCI base: procedures>+=
  subroutine mci_startup_message (mci, unit)
    class(mci_t), intent(in) :: mci
    integer, intent(in), optional :: unit
    if (mci%n_chain > 0) then
      write (msg_buffer, "(A,3(1x,I0,1x,A))" &
        "Integrator:", mci%n_chain, "chains,", &

```

```

        mci%n_channel, "channels,", &
        mci%n_dim, "dimensions"
    else
        write (msg_buffer, "(A,3(1x,I0,1x,A))") &
            "Integrator:", &
            mci%n_channel, "channels,", &
            mci%n_dim, "dimensions"
    end if
    call msg_message (unit = unit)
end subroutine mci_startup_message

```

There is no Initializer for the abstract type, but a generic setter for the number of channels and dimensions. We make two aliases available, to be able to override it.

```

<MCI base: mci: TBP>+=
    procedure :: set_dimensions => mci_set_dimensions
    procedure :: base_set_dimensions => mci_set_dimensions

<MCI base: procedures>+=
    subroutine mci_set_dimensions (mci, n_dim, n_channel)
        class(mci_t), intent(inout) :: mci
        integer, intent(in) :: n_dim
        integer, intent(in) :: n_channel
        mci%n_dim = n_dim
        mci%n_channel = n_channel
    end subroutine mci_set_dimensions

```

Declare particular dimensions as flat. This information can be used to simplify integration. When generating events, the flat dimensions should be sampled with uniform and uncorrelated distribution. It depends on the integrator what to do with that information.

```

<MCI base: mci: TBP>+=
    procedure (mci_declare_flat_dimensions), deferred :: declare_flat_dimensions

<MCI base: interfaces>=
    abstract interface
        subroutine mci_declare_flat_dimensions (mci, dim_flat)
            import
            class(mci_t), intent(inout) :: mci
            integer, dimension(:), intent(in) :: dim_flat
        end subroutine mci_declare_flat_dimensions
    end interface

```

Declare particular channels as equivalent, possibly allowing for permutations or reflections of dimensions. We use the information stored in the `phs_channel_t` object array that the phase-space module provides.

(We do not test this here, deferring the unit test to the `mci_vamp` implementation where we actually use this feature.)

```

<MCI base: mci: TBP>+=
    procedure (mci_declare_equivalences), deferred :: declare_equivalences

```

```

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_declare_equivalences (mci, channel, dim_offset)
      import
      class(mci_t), intent(inout) :: mci
      type(php_channel_t), dimension(:), intent(in) :: channel
      integer, intent(in) :: dim_offset
    end subroutine mci_declare_equivalences
  end interface

```

Declare particular channels as chained together. The implementation may use this array for keeping their weights equal to each other, etc.

The chain array is an array sized by the number of channels. For each channel, there is an integer entry that indicates the corresponding chains. The total number of chains is the maximum value of this entry.

```

<MCI base: mci: TBP>+≡
  procedure :: declare_chains => mci_declare_chains

<MCI base: procedures>+≡
  subroutine mci_declare_chains (mci, chain)
    class(mci_t), intent(inout) :: mci
    integer, dimension(:), intent(in) :: chain
    allocate (mci%chain (size (chain)))
    mci%n_chain = maxval (chain)
    mci%chain = chain
  end subroutine mci_declare_chains

```

Set the MD5 sum, independent of initialization.

```

<MCI base: mci: TBP>+≡
  procedure :: set_md5sum => mci_set_md5sum

<MCI base: procedures>+≡
  subroutine mci_set_md5sum (mci, md5sum)
    class(mci_t), intent(inout) :: mci
    character(32), intent(in) :: md5sum
    mci%md5sum = md5sum
  end subroutine mci_set_md5sum

```

Initialize a new integration pass. This is not necessarily meaningful, so we provide an empty base method. The mci\_vamp implementation overrides this.

```

<MCI base: mci: TBP>+≡
  procedure :: add_pass => mci_add_pass

<MCI base: procedures>+≡
  subroutine mci_add_pass (mci, adapt_grids, adapt_weights)
    class(mci_t), intent(inout) :: mci
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
  end subroutine mci_add_pass

```

Allocate an instance with matching type. This must be deferred.

```

<MCI base: mci: TBP>+≡
  procedure (mci_allocate_instance), deferred :: allocate_instance

```

```

<MCI base: interfaces>+=
  abstract interface
    subroutine mci_allocate_instance (mci, mci_instance)
      import
      class(mci_t), intent(in) :: mci
      class(mci_instance_t), intent(out), pointer :: mci_instance
    end subroutine mci_allocate_instance
  end interface

```

Import a random-number generator. We transfer the allocation of an existing generator state into the object. The generator state may already be initialized, or we can reset it by its `init` method.

```

<MCI base: mci: TBP>+=
  procedure :: import_rng => mci_import_rng

<MCI base: procedures>+=
  subroutine mci_import_rng (mci, rng)
    class(mci_t), intent(inout) :: mci
    class(rng_t), intent(inout), allocatable :: rng
    call move_alloc (rng, mci%rng)
  end subroutine mci_import_rng

```

Integrate: this depends on the implementation.

```

<MCI base: mci: TBP>+=
  procedure (mci_integrate), deferred :: integrate

<MCI base: interfaces>+=
  abstract interface
    subroutine mci_integrate (mci, instance, sampler, n_it, n_calls, results)
      import
      class(mci_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      class(mci_results_t), intent(inout), optional :: results
    end subroutine mci_integrate
  end interface

```

Analogously: event generation. Depending on the implementation, event generation may or may not require a previous integration pass.

Instead of a black-box `simulate` method, we require an initializer, a finalizer, and procedures for generating a single event. This allows us to interface simulation event by event from the outside, and it facilitates the further processing of an event after successful generation. For integration, this is not necessary.

The initializer has `intent(inout)` for the `mci` passed object. The reason is that the initializer can read integration results and grids from file, where the results can modify the `mci` record.

```

<MCI base: mci: TBP>+=
  procedure (mci_init_simulation), deferred :: init_simulation
  procedure (mci_final_simulation), deferred :: final_simulation

```



```

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_init_simulation (mci, instance)
      import
      class(mci_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
    end subroutine mci_init_simulation
  end interface

  abstract interface
    subroutine mci_final_simulation (mci, instance)
      import
      class(mci_t), intent(in) :: mci
      class(mci_instance_t), intent(inout) :: instance
    end subroutine mci_final_simulation
  end interface

```

The generated event will reside in the **instance** object (overall results and weight) and in the **sampler** object (detailed data). In the real application, we can subsequently call methods of the **sampler** in order to further process the generated event.

The **target** attributes are required by the VAMP implementation, which uses pointers to refer to the instance and sampler objects from within the integration function.

```

<MCI base: mci: TBP>+≡
  procedure (mci_generate), deferred :: generate_weighted_event
  procedure (mci_generate), deferred :: generate_unweighted_event

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_generate (mci, instance, sampler)
      import
      class(mci_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout), target :: instance
      class(mci_sampler_t), intent(inout), target :: sampler
    end subroutine mci_generate
  end interface

```

This is analogous, but we rebuild the event from the information stored in **state** instead of generating it.

```

<MCI base: mci: TBP>+≡
  procedure (mci_rebuild), deferred :: rebuild_event

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_rebuild (mci, instance, sampler, state)
      import
      class(mci_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      class(mci_state_t), intent(in) :: state
    end subroutine mci_rebuild
  end interface

```

Return the value of the integral, error, and efficiency.

```
(MCI base: mci: TBP)+≡
  procedure :: get_integral => mci_get_integral
  procedure :: get_error => mci_get_error
  procedure :: get_efficiency => mci_get_efficiency

(MCI base: procedures)+≡
  function mci_get_integral (mci) result (integral)
    class(mci_t), intent(in) :: mci
    real(default) :: integral
    if (mci%integral_known) then
      integral = mci%integral
    else
      ! error?
      integral = 0
    end if
  end function mci_get_integral

  function mci_get_error (mci) result (error)
    class(mci_t), intent(in) :: mci
    real(default) :: error
    if (mci%error_known) then
      error = mci%error
    else
      ! error?
      error = 0
    end if
  end function mci_get_error

  function mci_get_efficiency (mci) result (efficiency)
    class(mci_t), intent(in) :: mci
    real(default) :: efficiency
    if (mci%efficiency_known) then
      efficiency = mci%efficiency
    else
      ! error?
      efficiency = 0
    end if
  end function mci_get_efficiency
```

Return the MD5 sum of the configuration. This may be overridden in an extension, to return a different MD5 sum.

```
(MCI base: mci: TBP)+≡
  procedure :: get_md5sum => mci_get_md5sum

(MCI base: procedures)+≡
  function mci_get_md5sum (mci) result (md5sum)
    class(mci_t), intent(in) :: mci
    character(32) :: md5sum
    md5sum = mci%md5sum
  end function mci_get_md5sum
```

### 13.1.2 MCI instance

The base type contains an array of channel weights. The value `mci_weight` is the combined MCI weight that corresponds to a particular sampling point.

For convenience, we also store the `x` and Jacobian values for this sampling point.

```
<MCI base: public>+≡
  public :: mci_instance_t

<MCI base: types>+≡
  type, abstract :: mci_instance_t
    real(default), dimension(:), allocatable :: w
    real(default), dimension(:), allocatable :: f
    real(default), dimension(:,:), allocatable :: x
    integer :: selected_channel = 0
    real(default) :: mci_weight = 0
    real(default) :: integrand = 0
  contains
    <MCI base: mci instance: TBP>
  end type mci_instance_t
```

Output: deferred

```
<MCI base: mci instance: TBP>≡
  procedure (mci_instance_write), deferred :: write

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_instance_write (object, unit)
      import
      class(mci_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine mci_instance_write
  end interface
```

A finalizer, just in case.

```
<MCI base: mci instance: TBP>+≡
  procedure (mci_instance_final), deferred :: final

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_instance_final (object)
      import
      class(mci_instance_t), intent(inout) :: object
    end subroutine mci_instance_final
  end interface
```

Init: basic initializer for the arrays, otherwise deferred. Assigning the `mci` object is also deferred, because it depends on the concrete type.

The weights are initialized with an uniform normalized value.

```
<MCI base: mci instance: TBP>+≡
  procedure (mci_instance_base_init), deferred :: init
  procedure :: base_init => mci_instance_base_init
```

```

<MCI base: procedures>+≡
  subroutine mci_instance_base_init (mci_instance, mci)
    class(mci_instance_t), intent(out) :: mci_instance
    class(mci_t), intent(in), target :: mci
    allocate (mci_instance%w (mci%n_channel))
    allocate (mci_instance%f (mci%n_channel))
    allocate (mci_instance%x (mci%n_dim, mci%n_channel))
    if (size (mci_instance%w) /= 0) then
      mci_instance%w = 1._default / size (mci_instance%w)
    end if
    mci_instance%f = 0
    mci_instance%x = 0
  end subroutine mci_instance_base_init

```

Compute the overall weight factor for a configuration of  $x$  values and Jacobians  $f$ . The  $x$  values come in `n_channel` rows with `n_dim` entries each. The  $f$  factors constitute an array with `n_channel` entries.

We assume that the  $x$  and  $f$  arrays are already stored inside the MC instance. The result is also stored there.

```

<MCI base: mci instance: TBP>+≡
  procedure (mci_instance_compute_weight), deferred :: compute_weight

```

```

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_instance_compute_weight (mci, c)
      import
      class(mci_instance_t), intent(inout) :: mci
      integer, intent(in) :: c
    end subroutine mci_instance_compute_weight
  end interface

```

Record the integrand as returned by the sampler. Depending on the implementation, this may merely copy the value, or do more complicated things.

We may need the MCI weight for the actual computations, so this should be called after the previous routine.

```

<MCI base: mci instance: TBP>+≡
  procedure (mci_instance_record_integrand), deferred :: record_integrand

```

```

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_instance_record_integrand (mci, integrand)
      import
      class(mci_instance_t), intent(inout) :: mci
      real(default), intent(in) :: integrand
    end subroutine mci_instance_record_integrand
  end interface

```

Sample a point directly: evaluate the sampler, then compute the weight and the weighted integrand. Finally, record the integrand within the MCI instance.

```

<MCI base: mci instance: TBP>+≡
  procedure :: evaluate => mci_instance_evaluate

```

```

<MCI base: procedures>+=
  subroutine mci_instance_evaluate (mci, sampler, c, x)
    class(mci_instance_t), intent(inout) :: mci
    class(mci_sampler_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x
    real(default) :: val
    call sampler%evaluate (c, x, val, mci%x, mci%f)
    call mci%compute_weight (c)
    call mci%record_integrand (val)
  end subroutine mci_instance_evaluate

```

Assuming that the sampler is in a completely defined state, just extract the data that `evaluate` would compute. Also record the integrand.

```

<MCI base: mci instance: TBP>+=
  procedure :: fetch => mci_instance_fetch

<MCI base: procedures>+=
  subroutine mci_instance_fetch (mci, sampler, c)
    class(mci_instance_t), intent(inout) :: mci
    class(mci_sampler_t), intent(in) :: sampler
    integer, intent(in) :: c
    real(default) :: val
    call sampler%fetch (val, mci%x, mci%f)
    call mci%compute_weight (c)
    call mci%record_integrand (val)
  end subroutine mci_instance_fetch

```

The value, i.e., the weighted integrand, is the integrand (which should be taken as-is from the sampler) multiplied by the MCI weight.

```

<MCI base: mci instance: TBP>+=
  procedure :: get_value => mci_instance_get_value

<MCI base: procedures>+=
  function mci_instance_get_value (mci) result (value)
    class(mci_instance_t), intent(in) :: mci
    real(default) :: value
    value = mci%integrand * mci%weight
  end function mci_instance_get_value

```

This is an extra routine. By default, the event weight is equal to the value returned by the previous routine. However, if we select a channel for event generation not just based on the channel weights, the event weight has to account for this bias, so the event weight that applies to event generation is different. In that case, we should override the default routine.

```

<MCI base: mci instance: TBP>+=
  procedure :: get_event_weight => mci_instance_get_value

```

### 13.1.3 MCI state

This object can hold the relevant information that allows us to reconstruct the MCI instance without re-evaluating the sampler completely.

We store the `x_in` MC input parameter set, which coincides with the section of the complete `x` array that belongs to a particular channel. We also store the MC function value. When we want to reconstruct the state, we can use the input array to recover the complete `x` and `f` arrays (i.e., the kinematics), but do not need to recompute the MC function value (the dynamics).

The `mci_state_t` may be extended, to allow storing/recalling more information. In that case, we would override the type-bound procedures. However, the base type is also a concrete type and self-contained.

```

<MCI base: public>+≡
  public :: mci_state_t

<MCI base: types>+≡
  type :: mci_state_t
    integer :: selected_channel = 0
    real(default), dimension(:), allocatable :: x_in
    real(default) :: val
  contains
    <MCI base: mci state: TBP>
  end type mci_state_t

```

Output:

```

<MCI base: mci state: TBP>≡
  procedure :: write => mci_state_write

<MCI base: procedures>+≡
  subroutine mci_state_write (object, unit)
    class(mci_state_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "MCI state:"
    write (u, "(3x,A,I0)") "Channel   = ", object%selected_channel
    write (u, "(3x,A,999(1x,F12.10))") "x (in)   = ", object%x_in
    write (u, "(3x,A,ES19.12)") "Integrand = ", object%val
  end subroutine mci_state_write

```

To store the object, we take the relevant section of the `x` array. The channel used for storing data is taken from the `instance` object, but it could be arbitrary in principle.

```

<MCI base: mci instance: TBP>+≡
  procedure :: store => mci_instance_store

<MCI base: procedures>+≡
  subroutine mci_instance_store (mci, state)
    class(mci_instance_t), intent(in) :: mci
    class(mci_state_t), intent(out) :: state
    state%selected_channel = mci%selected_channel
    allocate (state%x_in (size (mci%x, 1)))
    state%x_in = mci%x(:,mci%selected_channel)
    state%val = mci%integrand
  end subroutine mci_instance_store

```

Recalling the state, we must consult the sampler in order to fully reconstruct the  $\mathbf{x}$  and  $\mathbf{f}$  arrays. The integrand value is known, and we also give it to the sampler, bypassing evaluation.

The final steps are equivalent to the `evaluate` method above.

```

<MCI base: mci instance: TBP>+≡
  procedure :: recall => mci_instance_recall

<MCI base: procedures>+≡
  subroutine mci_instance_recall (mci, sampler, state)
    class(mci_instance_t), intent(inout) :: mci
    class(mci_sampler_t), intent(inout) :: sampler
    class(mci_state_t), intent(in) :: state
    if (size (state%x_in) == size (mci%x, 1) &
        .and. state%selected_channel <= size (mci%x, 2)) then
      call sampler%rebuild (state%selected_channel, &
        state%x_in, state%val, mci%x, mci%f)
      call mci%compute_weight (state%selected_channel)
      call mci%record_integrand (state%val)
    else
      call msg_fatal ("Recalling event: mismatch in channel or dimension")
    end if
  end subroutine mci_instance_recall

```

### 13.1.4 MCI sampler

A sampler is an object that implements a multi-channel parameterization of the unit hypercube. Specifically, it is able to compute, given a channel and a set of  $x$  MC parameter values, a the complete set of  $x$  values and associated Jacobian factors  $f$  for all channels.

Furthermore, the sampler should return a single real value, the integrand, for the given point in the hypercube.

It must implement a method `evaluate` for performing the above computations.

```

<MCI base: public>+≡
  public :: mci_sampler_t

<MCI base: types>+≡
  type, abstract :: mci_sampler_t
  contains
    <MCI base: mci sampler: TBP>
  end type mci_sampler_t

```

Output, deferred to the implementation.

```

<MCI base: mci sampler: TBP>≡
  procedure (mci_sampler_write), deferred :: write

<MCI base: interfaces>+≡
  abstract interface
    subroutine mci_sampler_write (object, unit)
      import
      class(mci_sampler_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine mci_sampler_write
  end interface

```

```

        end subroutine mci_sampler_write
    end interface

```

The evaluation routine. Input is the channel index `c` and the one-dimensional parameter array `x_in`. Output are the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

```

<MCI base: mci_sampler: TBP>+≡
    procedure (mci_sampler_evaluate), deferred :: evaluate

<MCI base: interfaces>+≡
    abstract interface
        subroutine mci_sampler_evaluate (sampler, c, x_in, val, x, f)
            import
            class(mci_sampler_t), intent(inout) :: sampler
            integer, intent(in) :: c
            real(default), dimension(:), intent(in) :: x_in
            real(default), intent(out) :: val
            real(default), dimension(:,:), intent(out) :: x
            real(default), dimension(:), intent(out) :: f
        end subroutine mci_sampler_evaluate
    end interface

```

The shortcut. Again, the channel index `c` and the parameter array `x_in` are input. However, we also provide the integrand value `val`, and we just require that the complete parameter array `x` and Jacobian array `f` are recovered.

```

<MCI base: mci_sampler: TBP>+≡
    procedure (mci_sampler_rebuild), deferred :: rebuild

<MCI base: interfaces>+≡
    abstract interface
        subroutine mci_sampler_rebuild (sampler, c, x_in, val, x, f)
            import
            class(mci_sampler_t), intent(inout) :: sampler
            integer, intent(in) :: c
            real(default), dimension(:), intent(in) :: x_in
            real(default), intent(in) :: val
            real(default), dimension(:,:), intent(out) :: x
            real(default), dimension(:), intent(out) :: f
        end subroutine mci_sampler_rebuild
    end interface

```

This routine should extract the important data from a sampler that has been filled by other means. We fetch the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

```

<MCI base: mci_sampler: TBP>+≡
    procedure (mci_sampler_fetch), deferred :: fetch

<MCI base: interfaces>+≡
    abstract interface
        subroutine mci_sampler_fetch (sampler, val, x, f)
            import
            class(mci_sampler_t), intent(in) :: sampler
            real(default), intent(out) :: val

```



```

        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
    end subroutine mci_sampler_fetch
end interface

```

### 13.1.5 Results record

This is an abstract type which allows us to implement callback: each integration results can optionally be recorded to an instance of this object. The actual object may store a new result, average results, etc. It may also display a result on-line or otherwise, whenever the `record` method is called.

```

<MCI base: public>+≡
    public :: mci_results_t
<MCI base: types>+≡
    type, abstract :: mci_results_t
    contains
    <MCI base: mci results: TBP>
end type mci_results_t

```

The output routine is deferred. We provide an extra `verbose` flag, which could serve any purpose.

```

<MCI base: mci results: TBP>≡
    procedure (mci_results_write), deferred :: write
<MCI base: interfaces>+≡
    abstract interface
        subroutine mci_results_write (object, unit, verbose)
            import
            class(mci_results_t), intent(in) :: object
            integer, intent(in), optional :: unit
            logical, intent(in), optional :: verbose
        end subroutine mci_results_write
    end interface

```

This is the `record` method, which can be called directly from the integrator.

```

<MCI base: mci results: TBP>+≡
    procedure (mci_results_record), deferred :: record
<MCI base: interfaces>+≡
    abstract interface
        subroutine mci_results_record &
            (object, n_it, n_calls, integral, error, efficiency)
            import
            class(mci_results_t), intent(inout) :: object
            integer, intent(in) :: n_it
            integer, intent(in) :: n_calls
            real(default), intent(in) :: integral
            real(default), intent(in) :: error
            real(default), intent(in) :: efficiency
        end subroutine mci_results_record
    end interface

```

### 13.1.6 Unit tests

```
<MCI base: public>+≡
  public :: mci_base_test

<MCI base: tests>≡
  subroutine mci_base_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <MCI base: execute tests>
  end subroutine mci_base_test
```

#### Test implementation of the configuration type

The concrete type contains the number of requested calls and the integral result, to be determined.

The `max_factor` entry is set for the actual test integration, where the integrand is not unity but some other constant value. This value should be set here, such that the actual maximum of the integrand is known when vetoing unweighted events.

```
<MCI base: public>+≡
  public :: mci_test_t

<MCI base: test types>≡
  type, extends (mci_t) :: mci_test_t
    integer :: divisions = 0
    integer :: tries = 0
    real(default) :: max_factor = 1
  contains
    procedure :: final => mci_test_final
    procedure :: write => mci_test_write
    procedure :: startup_message => mci_test_startup_message
    procedure :: declare_flat_dimensions => mci_test_ignore_flat_dimensions
    procedure :: declare_equivalences => mci_test_ignore_equivalences
    procedure :: set_divisions => mci_test_set_divisions
    procedure :: set_max_factor => mci_test_set_max_factor
    procedure :: allocate_instance => mci_test_allocate_instance
    procedure :: integrate => mci_test_integrate
    procedure :: init_simulation => mci_test_ignore_init_simulation
    procedure :: final_simulation => mci_test_ignore_final_simulation
    procedure :: generate_weighted_event => mci_test_generate_weighted_event
    procedure :: generate_unweighted_event => &
      mci_test_generate_unweighted_event
    procedure :: rebuild_event => mci_test_rebuild_event
  end type mci_test_t
```

Finalizer: base version is sufficient

```
<MCI base: tests>+≡
  subroutine mci_test_final (object)
    class(mci_test_t), intent(inout) :: object
    call object%base_final ()
  end subroutine mci_test_final
```

Output: trivial

```
<MCI base: tests>+≡
subroutine mci_test_write (object, unit)
  class(mci_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)") "Test integrator:"
  call object%base_write (u)
  if (object%divisions /= 0) then
    write (u, "(3x,A,I0)") "Number of divisions = ", object%divisions
  end if
  if (allocated (object%rng)) call object%rng%write (u)
end subroutine mci_test_write
```

Short version.

```
<MCI base: tests>+≡
subroutine mci_test_startup_message (mci, unit)
  class(mci_test_t), intent(in) :: mci
  integer, intent(in), optional :: unit
  call mci%base_startup_message (unit)
  write (msg_buffer, "(A,1x,I0,1x,A)") &
    "Integrator: Test:", mci%divisions, "divisions"
  call msg_message (unit = unit)
end subroutine mci_test_startup_message
```

This is a no-op for the test integrator.

```
<MCI base: tests>+≡
subroutine mci_test_ignore_flat_dimensions (mci, dim_flat)
  class(mci_test_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
end subroutine mci_test_ignore_flat_dimensions
```

Ditto.

```
<MCI base: tests>+≡
subroutine mci_test_ignore_equivalences (mci, channel, dim_offset)
  class(mci_test_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
  integer, intent(in) :: dim_offset
end subroutine mci_test_ignore_equivalences
```

Set the number of divisions to a nonzero value.

```
<MCI base: tests>+≡
subroutine mci_test_set_divisions (object, divisions)
  class(mci_test_t), intent(inout) :: object
  integer, intent(in) :: divisions
  object%divisions = divisions
end subroutine mci_test_set_divisions
```

Set the maximum factor (default is 1).

```
<MCI base: tests>+≡
```

```

subroutine mci_test_set_max_factor (object, max_factor)
  class(mci_test_t), intent(inout) :: object
  real(default), intent(in) :: max_factor
  object%max_factor = max_factor
end subroutine mci_test_set_max_factor

```

Allocate instance with matching type.

```

<MCI base: tests>+≡
subroutine mci_test_allocate_instance (mci, mci_instance)
  class(mci_test_t), intent(in) :: mci
  class(mci_instance_t), intent(out), pointer :: mci_instance
  allocate (mci_test_instance_t :: mci_instance)
end subroutine mci_test_allocate_instance

```

Integrate: sample at the midpoints of uniform bits and add the results. We implement this for one and for two dimensions. In the latter case, we scan over two channels and multiply with the channel weights.

The arguments `n_it` and `n_calls` are ignored in this implementations.

The test integrator does not set error or efficiency, so those will remain undefined.

```

<MCI base: tests>+≡
subroutine mci_test_integrate (mci, instance, sampler, n_it, n_calls, results)
  class(mci_test_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  class(mci_results_t), intent(inout), optional :: results
  real(default), dimension(:), allocatable :: integral
  real(default), dimension(:), allocatable :: x
  integer :: i, j, c
  select type (instance)
  type is (mci_test_instance_t)
    allocate (integral (mci%n_channel))
    integral = 0
    allocate (x (mci%n_dim))
    select case (mci%n_dim)
    case (1)
      do c = 1, mci%n_channel
        do i = 1, mci%divisions
          x(1) = (i - 0.5_default) / mci%divisions
          call instance%evaluate (sampler, c, x)
          integral(c) = integral(c) + instance%get_value ()
        end do
      end do
      mci%integral = dot_product (instance%w, integral) &
        / mci%divisions
      mci%integral_known = .true.
    case (2)
      do c = 1, mci%n_channel
        do i = 1, mci%divisions
          x(1) = (i - 0.5_default) / mci%divisions

```

```

        do j = 1, mci%divisions
            x(2) = (j - 0.5_default) / mci%divisions
            call instance%evaluate (sampler, c, x)
            integral(c) = integral(c) + instance%get_value ()
        end do
    end do
end do
mci%integral = dot_product (instance%w, integral) &
    / mci%divisions / mci%divisions
mci%integral_known = .true.
end select
if (present (results)) then
    call results%record (n_it, n_calls, &
        mci%integral, mci%error, &
        efficiency = 0._default)
end if
end select
end subroutine mci_test_integrate

```

Simulation initializer and finalizer: nothing to do here.

```

<MCI base: tests>+≡
subroutine mci_test_ignore_init_simulation (mci, instance)
    class(mci_test_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout) :: instance
end subroutine mci_test_ignore_init_simulation

subroutine mci_test_ignore_final_simulation (mci, instance)
    class(mci_test_t), intent(in) :: mci
    class(mci_instance_t), intent(inout) :: instance
end subroutine mci_test_ignore_final_simulation

```

Event generator. We use mock random numbers for first selecting the channel and then setting the  $x$  values. The results reside in the state of **instance** and **sampler**.

```

<MCI base: tests>+≡
subroutine mci_test_generate_weighted_event (mci, instance, sampler)
    class(mci_test_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    real(default) :: r
    real(default), dimension(:), allocatable :: x
    integer :: c
    select type (instance)
    type is (mci_test_instance_t)
        allocate (x (mci%n_dim))
        select case (mci%n_channel)
        case (1)
            c = 1
            call mci%rng%generate (x(1))
        case (2)
            call mci%rng%generate (r)
            if (r < instance%w(1)) then
                c = 1
            end if
        end select
    end select
end subroutine mci_test_generate_weighted_event

```

```

        else
            c = 2
        end if
        call mci%rng%generate (x)
    end select
    call instance%evaluate (sampler, c, x)
end select
end subroutine mci_test_generate_weighted_event

```

For unweighted events, we generate weighted events and apply a simple rejection step to the relative event weight, until an event passes.

(This might result in an endless loop if we happen to be in sync with the mock random generator cycle. Therefore, limit the number of tries.)

*(MCI base: tests)*+≡

```

subroutine mci_test_generate_unweighted_event (mci, instance, sampler)
    class(mci_test_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    real(default) :: r
    integer :: i
    select type (instance)
    type is (mci_test_instance_t)
        mci%tries = 0
        do i = 1, 10
            call mci%generate_weighted_event (instance, sampler)
            mci%tries = mci%tries + 1
            call mci%rng%generate (r)
            if (r < instance%rel_value) exit
        end do
    end select
end subroutine mci_test_generate_unweighted_event

```

Here, we rebuild the event from the state without consulting the rng.

*(MCI base: tests)*+≡

```

subroutine mci_test_rebuild_event (mci, instance, sampler, state)
    class(mci_test_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout) :: instance
    class(mci_sampler_t), intent(inout) :: sampler
    class(mci_state_t), intent(in) :: state
    select type (instance)
    type is (mci_test_instance_t)
        call instance%recall (sampler, state)
    end select
end subroutine mci_test_rebuild_event

```

### Instance of the test MCI type

This instance type simulates the VAMP approach. We implement the VAMP multi-channel formula, but keep the channel-specific probability functions  $g_i$  smooth and fixed. We also keep the weights fixed.

The setup is as follows: we have  $n$  mappings of the unit hypercube

$$x = x(x^{(k)}) \quad \text{where } x = (x_1, \dots). \quad (13.1)$$

The Jacobian factors are the determinants

$$f^{(k)}(x^{(k)}) = \left| \frac{\partial x}{\partial x^{(k)}} \right| \quad (13.2)$$

We introduce arbitrary probability functions

$$g^{(k)}(x^{(k)}) \quad \text{with} \quad \int dx^{(k)} g^{(k)}(x^{(k)}) = 1 \quad (13.3)$$

and weights

$$w_k \quad \text{with} \quad \sum_k w_k = 1 \quad (13.4)$$

and construct the joint probability function

$$g(x) = \sum_k w_k \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \quad (13.5)$$

which also satisfies

$$\int g(x) dx = 1. \quad (13.6)$$

The algorithm implements a resolution of unity as follows

$$\begin{aligned} 1 &= \int dx = \int \frac{g(x)}{g(x)} dx \\ &= \sum_k w_k \int \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \frac{dx}{g(x)} \\ &= \sum_k w_k \int g^{(k)}(x^{(k)}) \frac{dx^{(k)}}{g(x^{(k)})} \end{aligned} \quad (13.7)$$

where each of the integrals in the sum is evaluated using the channel-specific variables  $x^{(k)}$ .

We provide two examples: (1) trivial with one channel, one dimension, and all functions unity and (2) two channels and two dimensions with

$$\begin{aligned} x(x^{(1)}) &= (x_1^{(1)}, x_2^{(1)}) \\ x(x^{(2)}) &= (x_1^{(2)2}, x_2^{(2)}) \end{aligned} \quad (13.8)$$

hence

$$f^{(1)} \equiv 1, \quad f^{(2)}(x^{(2)}) = 2x_1^{(2)} \quad (13.9)$$

The probability functions are

$$g^{(1)} \equiv 1, \quad g^{(2)}(x^{(2)}) = 2x_2^{(2)} \quad (13.10)$$

In the concrete implementation of the integrator instance we store values for the channel probabilities  $g_i$  and the accumulated probability  $g$ .

We also store the result (product of integrand and MCI weight), the expected maximum for the result in each channel.

```
<MCI base: public>+≡
public :: mci_test_instance_t
```

```

<MCI base: test types>+≡
type, extends (mci_instance_t) :: mci_test_instance_t
  type(mci_test_t), pointer :: mci => null ()
  real(default) :: g = 0
  real(default), dimension(:), allocatable :: gi
  real(default) :: value = 0
  real(default) :: rel_value = 0
  real(default), dimension(:), allocatable :: max
contains
  procedure :: write => mci_test_instance_write
  procedure :: final => mci_test_instance_final
  procedure :: init => mci_test_instance_init
  procedure :: compute_weight => mci_test_instance_compute_weight
  procedure :: record_integrand => mci_test_instance_record_integrand
end type mci_test_instance_t

```

Output: trivial

```

<MCI base: tests>+≡
subroutine mci_test_instance_write (object, unit)
  class(mci_test_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, c
  u = output_unit (unit)
  write (u, "(1x,A,ES13.7)") "Result value = ", object%value
  write (u, "(1x,A,ES13.7)") "Rel. weight = ", object%rel_value
  write (u, "(1x,A,ES13.7)") "Integrand = ", object%integrand
  write (u, "(1x,A,ES13.7)") "MCI weight = ", object%mci_weight
  write (u, "(3x,A,I0)") "c = ", object%selected_channel
  write (u, "(3x,A,ES13.7)") "g = ", object%g
  write (u, "(1x,A)") "Channel parameters:"
  do c = 1, object%mci%n_channel
    write (u, "(1x,I0,A,4(1x,ES13.7))") c, ": w/f/g/m =", &
      object%w(c), object%f(c), object%gi(c), object%max(c)
    write (u, "(4x,A,9(1x,F9.7))") "x =", object%x(:,c)
  end do
end subroutine mci_test_instance_write

```

The finalizer is empty.

```

<MCI base: tests>+≡
subroutine mci_test_instance_final (object)
  class(mci_test_instance_t), intent(inout) :: object
end subroutine mci_test_instance_final

```

Initializer. We make use of the analytical result that the maximum of the weighted integrand, in each channel, is equal to 1 (one-dimensional case) and 2 (two-dimensional case), respectively.

```

<MCI base: tests>+≡
subroutine mci_test_instance_init (mci_instance, mci)
  class(mci_test_instance_t), intent(out) :: mci_instance
  class(mci_t), intent(in), target :: mci
  call mci_instance%base_init (mci)
  select type (mci)

```



```

type is (mci_test_t)
  mci_instance%mci => mci
end select
allocate (mci_instance%gi (mci%n_channel))
mci_instance%gi = 0
allocate (mci_instance%max (mci%n_channel))
select case (mci%n_channel)
case (1)
  mci_instance%max = 1._default
case (2)
  mci_instance%max = 2._default
end select
end subroutine mci_test_instance_init

```

Compute weight: we implement the VAMP multi-channel formula. The channel probabilities `gi` are predefined functions.

*(MCI base: tests)+≡*

```

subroutine mci_test_instance_compute_weight (mci, c)
  class(mci_test_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  integer :: i
  mci%selected_channel = c
  select case (mci%mci%n_dim)
  case (1)
    mci%gi(1) = 1
  case (2)
    mci%gi(1) = 1
    mci%gi(2) = 2 * mci%x(2,2)
  end select
  mci%g = 0
  do i = 1, mci%mci%n_channel
    mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
  end do
  mci%mci_weight = mci%gi(c) / mci%g
end subroutine mci_test_instance_compute_weight

```

Record the integrand. Apply the Jacobian weight to get the absolute value. Divide by the channel maximum and by any overall factor to get the value relative to the maximum.

*(MCI base: tests)+≡*

```

subroutine mci_test_instance_record_integrand (mci, integrand)
  class(mci_test_instance_t), intent(inout) :: mci
  real(default), intent(in) :: integrand
  mci%integrand = integrand
  mci%value = mci%integrand * mci%mci_weight
  mci%rel_value = mci%value / mci%max(mci%selected_channel) &
    / mci%mci%max_factor
end subroutine mci_test_instance_record_integrand

```

## Test sampler

The test sampler implements a fixed configuration, either trivial (one-channel, one-dimension), or slightly nontrivial (two-channel, two-dimension). In the second channel, the first parameter is mapped according to  $x_1 = x_1^{(2)2}$ , so we have  $f^{(2)}(x^{(2)}) = 2x_1^{(2)}$ .

For display purposes, we store the return values inside the object. This is not strictly necessary.

```
<MCI base: test types>+≡
type, extends (mci_sampler_t) :: test_sampler_t
  real(default) :: integrand = 0
  integer :: selected_channel = 0
  real(default), dimension(:,:), allocatable :: x
  real(default), dimension(:), allocatable :: f
contains
  procedure :: init => test_sampler_init
  procedure :: write => test_sampler_write
  procedure :: compute => test_sampler_compute
  procedure :: evaluate => test_sampler_evaluate
  procedure :: rebuild => test_sampler_rebuild
  procedure :: fetch => test_sampler_fetch
end type test_sampler_t
```

```
<MCI base: tests>+≡
subroutine test_sampler_init (sampler, n)
  class(test_sampler_t), intent(out) :: sampler
  integer, intent(in) :: n
  allocate (sampler%x (n, n))
  allocate (sampler%f (n))
end subroutine test_sampler_init
```

## Output

```
<MCI base: tests>+≡
subroutine test_sampler_write (object, unit)
  class(test_sampler_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, c
  u = output_unit (unit)
  write (u, "(1x,A)") "Test sampler:"
  write (u, "(3x,A,ES13.7)") "Integrand = ", object%integrand
  write (u, "(3x,A,I0)") "Channel = ", object%selected_channel
  do c = 1, size (object%f)
    write (u, "(1x,I0,':',1x,A,ES13.7)") c, "f = ", object%f(c)
    write (u, "(4x,A,9(1x,F9.7))") "x =", object%x(:,c)
  end do
end subroutine test_sampler_write
```

Compute  $x$  and Jacobians, given the input parameter array. This is called both by `evaluate` and `rebuild`.

```
<MCI base: tests>+≡
subroutine test_sampler_compute (sampler, c, x_in)
```

```

class(test_sampler_t), intent(inout) :: sampler
integer, intent(in) :: c
real(default), dimension(:), intent(in) :: x_in
sampler%selected_channel = c
select case (size (sampler%f))
case (1)
  sampler%x(:,1) = x_in
  sampler%f = 1
case (2)
  select case (c)
  case (1)
    sampler%x(:,1) = x_in
    sampler%x(1,2) = sqrt (x_in(1))
    sampler%x(2,2) = x_in(2)
  case (2)
    sampler%x(1,1) = x_in(1) ** 2
    sampler%x(2,1) = x_in(2)
    sampler%x(:,2) = x_in
  end select
  sampler%f(1) = 1
  sampler%f(2) = 2 * sampler%x(1,2)
end select
end subroutine test_sampler_compute

```

The integrand is always equal to 1.

*(MCI base: tests)*+≡

```

subroutine test_sampler_evaluate (sampler, c, x_in, val, x, f)
class(test_sampler_t), intent(inout) :: sampler
integer, intent(in) :: c
real(default), dimension(:), intent(in) :: x_in
real(default), intent(out) :: val
real(default), dimension(:,:), intent(out) :: x
real(default), dimension(:), intent(out) :: f
call sampler%compute (c, x_in)
sampler%integrand = 1
val = sampler%integrand
x = sampler%x
f = sampler%f
end subroutine test_sampler_evaluate

```

Construct kinematics from the input  $x$  array. Set the integrand instead of evaluating it.

*(MCI base: tests)*+≡

```

subroutine test_sampler_rebuild (sampler, c, x_in, val, x, f)
class(test_sampler_t), intent(inout) :: sampler
integer, intent(in) :: c
real(default), dimension(:), intent(in) :: x_in
real(default), intent(in) :: val
real(default), dimension(:,:), intent(out) :: x
real(default), dimension(:), intent(out) :: f
call sampler%compute (c, x_in)
sampler%integrand = val
x = sampler%x

```

```

        f = sampler%f
    end subroutine test_sampler_rebuild

```

Recall contents.

```

<MCI base: tests>+≡
subroutine test_sampler_fetch (sampler, val, x, f)
    class(test_sampler_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%integrand
    x = sampler%x
    f = sampler%f
end subroutine test_sampler_fetch

```

### Test results object

This mock object just stores and displays the current result.

```

<MCI base: test types>+≡
type, extends (mci_results_t) :: mci_test_results_t
    integer :: n_it = 0
    integer :: n_calls = 0
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: efficiency = 0
contains
    <MCI base: mci test results: TBP>
end type mci_test_results_t

```

Output.

```

<MCI base: mci test results: TBP>≡
    procedure :: write => mci_test_results_write

<MCI base: tests>+≡
subroutine mci_test_results_write (object, unit, verbose)
    class(mci_test_results_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A,1x,I0)") "Iterations = ", object%n_it
    write (u, "(3x,A,1x,I0)") "Calls      = ", object%n_calls
    write (u, "(3x,A,1x,F12.10)") "Integral  = ", object%integral
    write (u, "(3x,A,1x,F12.10)") "Error      = ", object%error
    write (u, "(3x,A,1x,F12.10)") "Efficiency = ", object%efficiency
end subroutine mci_test_results_write

```

Record result.

```

<MCI base: mci test results: TBP>+≡
    procedure :: record => mci_test_results_record

```

```

<MCI base: tests>+≡
  subroutine mci_test_results_record &
    (object, n_it, n_calls, integral, error, efficiency)
    class(mci_test_results_t), intent(inout) :: object
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    real(default), intent(in) :: integral
    real(default), intent(in) :: error
    real(default), intent(in) :: efficiency
    object%n_it = n_it
    object%n_calls = n_calls
    object%integral = integral
    object%error = error
    object%efficiency = efficiency
  end subroutine mci_test_results_record

```

### Integrator configuration data

Construct and display a test integrator configuration object.

```

<MCI base: execute tests>≡
  call test (mci_base_1, "mci_base_1", &
    "integrator configuration", &
    u, results)

<MCI base: tests>+≡
  subroutine mci_base_1 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler

    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_base_1"
    write (u, "(A)")  "* Purpose: initialize and display &
      &test integrator"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

```

```

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
    call sampler%init (2)
end select

write (u, "(A)")  "* Evaluate sampler for given point and channel"
write (u, "(A)")

call sampler%evaluate (1, [0.25_default, 0.8_default], &
    integrand, mci_instance%x, mci_instance%f)

call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute MCI weight"
write (u, "(A)")

call mci_instance%compute_weight (1)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Get integrand and compute weight for another point"
write (u, "(A)")

call mci_instance%evaluate (sampler, 2, [0.5_default, 0.6_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recall results, again"
write (u, "(A)")

call mci_instance%final ()
deallocate (mci_instance)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

call mci_instance%fetch (sampler, 2)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Retrieve value"
write (u, "(A)")

write (u, "(1x,A,ES13.7)")  "Weighted integrand = ", &
    mci_instance%get_value ()

call mci_instance%final ()
call mci%final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_1"

end subroutine mci_base_1

```

### Trivial integral

Use the MCI approach to compute a trivial one-dimensional integral.

```

⟨MCI base: execute tests⟩+≡
  call test (mci_base_2, "mci_base_2", &
    "integration", &
    u, results)
⟨MCI base: tests⟩+≡
  subroutine mci_base_2 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler

    write (u, "(A)")  "* Test output: mci_base_2"
    write (u, "(A)")  "* Purpose: perform a test integral"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (1, 1)
    select type (mci)
    type is (mci_test_t)
      call mci%set_divisions (10)
    end select

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_t :: sampler)
    select type (sampler)
    type is (test_sampler_t)
      call sampler%init (1)
    end select

    write (u, "(A)")  "* Integrate"

```

```

write (u, "(A)")

call mci%integrate (mci_instance, sampler, 0, 0)

call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_2"

end subroutine mci_base_2

```

### Nontrivial integral

Use the MCI approach to compute a simple two-dimensional integral with two channels.

```

<MCI base: execute tests>+≡
  call test (mci_base_3, "mci_base_3", &
    "integration (two channels)", &
    u, results)
<MCI base: tests>+≡
  subroutine mci_base_3 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler

    write (u, "(A)")  "* Test output: mci_base_3"
    write (u, "(A)")  "* Purpose: perform a nontrivial test integral"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_test_t)
      call mci%set_divisions (10)
    end select

    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

```



```

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
    call sampler%init (2)
end select

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call mci%integrate (mci_instance, sampler, 0, 0)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with higher resolution"
write (u, "(A)")

select type (mci)
type is (mci_test_t)
    call mci%set_divisions (100)
end select

call mci%integrate (mci_instance, sampler, 0, 0)
call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_3"

end subroutine mci_base_3

```

## Event generation

We generate “random” events, one weighted and one unweighted. The test implementation does not require an integration pass, we can generate events immediately.

```

<MCI base: execute tests>+≡
    call test (mci_base_4, "mci_base_4", &
        "event generation (two channels)", &
        u, results)

<MCI base: tests>+≡
subroutine mci_base_4 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_base_4"
    write (u, "(A)")  "* Purpose: generate events"
    write (u, "(A)")

```

```

write (u, "(A)")  "* Initialize integrator, instance, sampler"
write (u, "(A)")

allocate (mci_test_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_test_t)
    call mci%set_divisions (10)
end select

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
    call sampler%init (2)
end select

allocate (rng_test_t :: rng)
call mci%import_rng (rng)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)

call sampler%write (u)
write (u, *)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call mci%generate_unweighted_event (mci_instance, sampler)

select type (mci)
type is (mci_test_t)
    write (u, "(A,I0)")  " Success in try ", mci%tries
    write (u, "(A)")
end select

call sampler%write (u)
write (u, *)
call mci_instance%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_4"

```

```
end subroutine mci_base_4
```

### Store and recall data

We generate an event and store the relevant data, i.e., the input parameters and the result value for a particular channel. Then we use those data to recover the event, as far as the MCI record is concerned.

```
(MCI base: execute tests)+≡
  call test (mci_base_5, "mci_base_5", &
    "store and recall", &
    u, results)

(MCI base: tests)+≡
  subroutine mci_base_5 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    class(mci_state_t), allocatable :: state

    write (u, "(A)")  "* Test output: mci_base_5"
    write (u, "(A)")  "* Purpose: store and recall an event"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator, instance, sampler"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_test_t)
      call mci%set_divisions (10)
    end select

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_t :: sampler)
    select type (sampler)
    type is (test_sampler_t)
      call sampler%init (2)
    end select

    allocate (rng_test_t :: rng)
    call mci%import_rng (rng)

    write (u, "(A)")  "* Generate weighted event"
    write (u, "(A)")

    call mci%generate_weighted_event (mci_instance, sampler)

    call sampler%write (u)
```

```

write (u, *)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Store data"
write (u, "(A)")

allocate (state)
call mci_instance%store (state)
call mci_instance%final ()
deallocate (mci_instance)

call state%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recall data and rebuild event"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)
call mci%rebuild_event (mci_instance, sampler, state)

call sampler%write (u)
write (u, *)
call mci_instance%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_5"

end subroutine mci_base_5

```

## Chained channels

Chain channels together. In the base configuration, this just fills entries in an extra array (each channel may belong to a chain). In type implementations, this will be used for grouping equivalent channels by keeping their weights equal.

*(MCI base: execute tests)*+≡

```

call test (mci_base_6, "mci_base_6", &
  "chained channels", &
  u, results)

```

*(MCI base: tests)*+≡

```

subroutine mci_base_6 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci

  write (u, "(A)")  "* Test output: mci_base_6"
  write (u, "(A)")  "* Purpose: initialize and display &
    &test integrator with chains"
  write (u, "(A)")

```

```

write (u, "(A)")  "* Initialize integrator"
write (u, "(A)")

allocate (mci_test_t :: mci)
call mci%set_dimensions (1, 5)

write (u, "(A)")  "* Introduce chains"
write (u, "(A)")

call mci%declare_chains ([1, 2, 2, 1, 2])

call mci%write (u)

call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_6"

end subroutine mci_base_6

```

## Recording results

Compute a simple two-dimensional integral and record the result.

*(MCI base: execute tests)+≡*

```

call test (mci_base_7, "mci_base_7", &
  "recording results", &
  u, results)

```

*(MCI base: tests)+≡*

```

subroutine mci_base_7 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(mci_results_t), allocatable :: results

  write (u, "(A)")  "* Test output: mci_base_7"
  write (u, "(A)")  "* Purpose: perform a nontrivial test integral &
    &and record results"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_test_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_test_t)
    call mci%set_divisions (10)
  end select

  write (u, "(A)")  "* Initialize instance"

```

```

write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
  call sampler%init (2)
end select

allocate (mci_test_results_t :: results)

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call mci%integrate (mci_instance, sampler, 1, 1000, results)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Display results"
write (u, "(A)")

call results%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_base_7"

end subroutine mci_base_7

```

## 13.2 Simple midpoint integration

This is a most simple implementation of an integrator. The algorithm is the straightforward multi-dimensional midpoint rule, i.e., the integration hypercube is binned uniformly, the integrand is evaluated at the midpoints of each bin, and the result is the average. The binning is equivalent for all integration dimensions.

This rule is accurate to the order  $h^2$ , where  $h$  is the bin width. Given that  $h = N^{-1/d}$ , where  $d$  is the integration dimension and  $N$  is the total number of sampling points, we get a relative error of order  $N^{-2/d}$ . This is superior to MC integration if  $d < 4$ , and equivalent if  $d = 4$ . It is not worse than higher-order formulas (such as Gauss integration) if the integrand is not smooth, e.g., if it contains cuts.

The integrator is specifically single-channel. However, we do not limit the dimension.

`<mci_midpoint.f90>≡`

```

<File header>

module mci_midpoint

    use kinds !NODEP!
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use unit_tests

    use phs_base
    use rng_base
    use mci_base

    <Standard module head>

    <MCI midpoint: public>

    <MCI midpoint: types>

    <MCI midpoint: interfaces>

    <MCI midpoint: test types>

contains

    <MCI midpoint: procedures>

    <MCI midpoint: tests>

end module mci_midpoint

```

### 13.2.1 Integrator

The object contains the methods for integration and event generation. For the actual work and data storage, it spawns an instance object.

After an integration pass, we update the `max` parameter to indicate the maximum absolute value of the integrand that the integrator encountered. This is required for event generation.

```

<MCI midpoint: public>≡
    public :: mci_midpoint_t

<MCI midpoint: types>≡
    type, extends (mci_t) :: mci_midpoint_t
        integer :: n_dim_binned = 0
        logical, dimension(:), allocatable :: dim_is_binned
        logical :: calls_known = .false.
        integer :: n_calls = 0
        integer :: n_calls_pos = 0
        integer :: n_calls_nul = 0
        integer :: n_calls_neg = 0
        real(default) :: integral_pos = 0
        real(default) :: integral_neg = 0
        integer, dimension(:), allocatable :: n_bin

```

```

        logical :: max_known = .false.
        real(default) :: max = 0
        real(default) :: min = 0
        real(default) :: max_abs = 0
        real(default) :: min_abs = 0
    contains
        <MCI midpoint: mci midpoint: TBP>
    end type mci_midpoint_t

```

Finalizer: base version is sufficient

```

<MCI midpoint: mci midpoint: TBP>≡
    procedure :: final => mci_midpoint_final

<MCI midpoint: procedures>≡
    subroutine mci_midpoint_final (object)
        class(mci_midpoint_t), intent(inout) :: object
        call object%base_final ()
    end subroutine mci_midpoint_final

```

Output.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: write => mci_midpoint_write

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_write (object, unit)
        class(mci_midpoint_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
        write (u, "(1x,A)") "Single-channel midpoint rule integrator:"
        call object%base_write (u)
        if (object%n_dim_binned < object%n_dim) then
            write (u, "(3x,A,99(1x,I0))") "Flat dimensions      =", &
                pack ([i, i = 1, object%n_dim], mask = .not. object%dim_is_binned)
            write (u, "(3x,A,I0)") "Number of binned dim = ", object%n_dim_binned
        end if
        if (object%calls_known) then
            write (u, "(3x,A,99(1x,I0))") "Number of bins      =", object%n_bin
            write (u, "(3x,A,I0)") "Number of calls     =", object%n_calls
            if (object%n_calls_pos /= object%n_calls) then
                write (u, "(3x,A,I0)") "  positive value    =", object%n_calls_pos
                write (u, "(3x,A,I0)") "  zero value        =", object%n_calls_nul
                write (u, "(3x,A,I0)") "  negative value     =", object%n_calls_neg
                write (u, "(3x,A,ES17.10)") &
                    "Integral (pos. part) = ", object%integral_pos
                write (u, "(3x,A,ES17.10)") &
                    "Integral (neg. part) = ", object%integral_neg
            end if
        end if
        if (object%max_known) then
            write (u, "(3x,A,ES17.10)") "Maximum of integrand = ", object%max
            write (u, "(3x,A,ES17.10)") "Minimum of integrand = ", object%min
            if (object%min /= object%min_abs) then
                write (u, "(3x,A,ES17.10)") "Maximum (abs. value) = ", object%max_abs
            end if
        end if
    end subroutine mci_midpoint_write

```



```

        write (u, "(3x,A,ES17.10)") "Minimum (abs. value) = ", object%min_abs
    end if
end if
    if (allocated (object%rng)) call object%rng%write (u)
end subroutine mci_midpoint_write

```

Startup message: short version.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: startup_message => mci_midpoint_startup_message

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_startup_message (mci, unit)
        class(mci_midpoint_t), intent(in) :: mci
        integer, intent(in), optional :: unit
        call mci%base_startup_message (unit)
        if (mci%n_dim_binned < mci%n_dim) then
            write (msg_buffer, "(A,2(1x,I0,1x,A))" ) &
                "Integrator: Midpoint rule:", &
                mci%n_dim_binned, "binned dimensions"
        else
            write (msg_buffer, "(A,2(1x,I0,1x,A))" ) &
                "Integrator: Midpoint rule"
        end if
        call msg_message (unit = unit)
    end subroutine mci_midpoint_startup_message

```

The number of channels must be one.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: set_dimensions => mci_midpoint_set_dimensions

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_set_dimensions (mci, n_dim, n_channel)
        class(mci_midpoint_t), intent(inout) :: mci
        integer, intent(in) :: n_dim
        integer, intent(in) :: n_channel
        if (n_channel == 1) then
            mci%n_channel = n_channel
            mci%n_dim = n_dim
            allocate (mci%dim_is_binned (mci%n_dim))
            mci%dim_is_binned = .true.
            mci%n_dim_binned = count (mci%dim_is_binned)
            allocate (mci%n_bin (mci%n_dim))
            mci%n_bin = 0
        else
            call msg_fatal ("Attempt to initialize single-channel integrator &
                &for multiple channels")
        end if
    end subroutine mci_midpoint_set_dimensions

```

Declare particular dimensions as flat. These dimensions will not be binned.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: declare_flat_dimensions => mci_midpoint_declare_flat_dimensions

```

```

<MCI midpoint: procedures>+≡
subroutine mci_midpoint_declare_flat_dimensions (mci, dim_flat)
  class(mci_midpoint_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
  integer :: d
  mci%n_dim_binned = mci%n_dim - size (dim_flat)
  do d = 1, size (dim_flat)
    mci%dim_is_binned(dim_flat(d)) = .false.
  end do
  mci%n_dim_binned = count (mci%dim_is_binned)
end subroutine mci_midpoint_declare_flat_dimensions

```

Declare particular channels as equivalent. This has no effect.

```

<MCI midpoint: mci midpoint: TBP>+≡
  procedure :: declare_equivalences => mci_midpoint_ignore_equivalences
<MCI midpoint: procedures>+≡
subroutine mci_midpoint_ignore_equivalences (mci, channel, dim_offset)
  class(mci_midpoint_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
  integer, intent(in) :: dim_offset
end subroutine mci_midpoint_ignore_equivalences

```

Allocate instance with matching type.

```

<MCI midpoint: mci midpoint: TBP>+≡
  procedure :: allocate_instance => mci_midpoint_allocate_instance
<MCI midpoint: procedures>+≡
subroutine mci_midpoint_allocate_instance (mci, mci_instance)
  class(mci_midpoint_t), intent(in) :: mci
  class(mci_instance_t), intent(out), pointer :: mci_instance
  allocate (mci_midpoint_instance_t :: mci_instance)
end subroutine mci_midpoint_allocate_instance

```

Integrate. The number of dimensions is arbitrary. We make sure that the number of calls is evenly distributed among the dimensions. The actual number of calls will typically be smaller than the requested number, but never smaller than 1.

The sampling over a variable number of dimensions implies a variable number of nested loops. We implement this by a recursive subroutine, one loop in each recursion level.

The number of iterations `n_it` is ignored. Also, the error is set to zero in the current implementation.

```

<MCI midpoint: mci midpoint: TBP>+≡
  procedure :: integrate => mci_midpoint_integrate
<MCI midpoint: procedures>+≡
subroutine mci_midpoint_integrate (mci, instance, sampler, n_it, n_calls, &
  results)
  class(mci_midpoint_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: n_it

```

```

integer, intent(in) :: n_calls
class(mci_results_t), intent(inout), optional :: results
real(default), dimension(:), allocatable :: x
real(default) :: integral, integral_pos, integral_neg
integer :: n_bin, d
select type (instance)
type is (mci_midpoint_instance_t)
  allocate (x (mci%n_dim))
  integral = 0
  integral_pos = 0
  integral_neg = 0
  select case (mci%n_dim_binned)
  case (1)
    n_bin = n_calls
  case (2:)
    n_bin = max (int (n_calls ** (1. / mci%n_dim_binned)), 1)
  end select
  where (mci%dim_is_binned)
    mci%n_bin = n_bin
  elsewhere
    mci%n_bin = 1
  end where
  mci%n_calls = product (mci%n_bin)
  mci%n_calls_pos = 0
  mci%n_calls_nul = 0
  mci%n_calls_neg = 0
  mci%calls_known = .true.
  call sample_dim (mci%n_dim)
  mci%integral = integral / mci%n_calls
  mci%integral_pos = integral_pos / mci%n_calls
  mci%integral_neg = integral_neg / mci%n_calls
  mci%integral_known = .true.
  call instance%set_max ()
  if (present (results)) then
    call results%record (1, mci%n_calls, &
      mci%integral, mci%error, mci%efficiency)
  end if
end select
contains
recursive subroutine sample_dim (d)
  integer, intent(in) :: d
  integer :: i
  real(default) :: value
  do i = 1, mci%n_bin(d)
    x(d) = (i - 0.5_default) / mci%n_bin(d)
    if (d > 1) then
      call sample_dim (d - 1)
    else
      call instance%evaluate (sampler, 1, x)
      value = instance%get_value ()
      if (value > 0) then
        mci%n_calls_pos = mci%n_calls_pos + 1
        integral = integral + value
        integral_pos = integral_pos + value
      end if
    end if
  end do
end subroutine sample_dim

```

```

        else if (value == 0) then
            mci%n_calls_nul = mci%n_calls_nul + 1
        else
            mci%n_calls_neg = mci%n_calls_neg + 1
            integral = integral + value
            integral_neg = integral_neg + value
        end if
    end if
end do
end subroutine sample_dim
end subroutine mci_midpoint_integrate

```

Simulation initializer and finalizer: nothing to do here.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: init_simulation => mci_midpoint_ignore_init_simulation
    procedure :: final_simulation => mci_midpoint_ignore_final_simulation

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_ignore_init_simulation (mci, instance)
        class(mci_midpoint_t), intent(inout) :: mci
        class(mci_instance_t), intent(inout) :: instance
    end subroutine mci_midpoint_ignore_init_simulation

    subroutine mci_midpoint_ignore_final_simulation (mci, instance)
        class(mci_midpoint_t), intent(in) :: mci
        class(mci_instance_t), intent(inout) :: instance
    end subroutine mci_midpoint_ignore_final_simulation

```

Generate weighted event.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: generate_weighted_event => mci_midpoint_generate_weighted_event

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_generate_weighted_event (mci, instance, sampler)
        class(mci_midpoint_t), intent(inout) :: mci
        class(mci_instance_t), intent(inout), target :: instance
        class(mci_sampler_t), intent(inout), target :: sampler
        real(default), dimension(mci%n_dim) :: x
        call mci%rng%generate (x)
        call instance%evaluate (sampler, 1, x)
    end subroutine mci_midpoint_generate_weighted_event

```

For unweighted events, we generate weighted events and apply a simple rejection step to the relative event weight, until an event passes.

Note that we use the `max_abs` value stored in the configuration record, not the one stored in the instance. The latter may change during event generation. After an event generation pass is over, we may update the value for a subsequent pass.

```

<MCI midpoint: mci midpoint: TBP>+≡
    procedure :: generate_unweighted_event => &
        mci_midpoint_generate_unweighted_event

```

```

<MCI midpoint: procedures>+≡
  subroutine mci_midpoint_generate_unweighted_event (mci, instance, sampler)
    class(mci_midpoint_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    real(default) :: x
    select type (instance)
    type is (mci_midpoint_instance_t)
      if (mci%max_known .and. mci%max_abs > 0) then
        do
          call mci%generate_weighted_event (instance, sampler)
          call mci%rng%generate (x)
          if (x * mci%max_abs <= abs (instance%integrand)) exit
        end do
      else
        call msg_fatal ("Unweighted event generation: &
          &maximum of integrand is zero or unknown")
      end if
    end select
  end subroutine mci_midpoint_generate_unweighted_event

```

Rebuild an event, using the state input.

```

<MCI midpoint: mci midpoint: TBP>+≡
  procedure :: rebuild_event => mci_midpoint_rebuild_event

<MCI midpoint: procedures>+≡
  subroutine mci_midpoint_rebuild_event (mci, instance, sampler, state)
    class(mci_midpoint_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout) :: instance
    class(mci_sampler_t), intent(inout) :: sampler
    class(mci_state_t), intent(in) :: state
    select type (instance)
    type is (mci_midpoint_instance_t)
      call instance%recall (sampler, state)
    end select
  end subroutine mci_midpoint_rebuild_event

```

### 13.2.2 Integrator instance

Covering the case of flat dimensions, we store a complete x array. This is filled when generating events.

```

<MCI midpoint: public>+≡
  public :: mci_midpoint_instance_t

<MCI midpoint: types>+≡
  type, extends (mci_instance_t) :: mci_midpoint_instance_t
    type(mci_midpoint_t), pointer :: mci => null ()
    logical :: max_known = .false.
    real(default) :: max = 0
    real(default) :: min = 0
    real(default) :: max_abs = 0
    real(default) :: min_abs = 0
  contains

```

```

    <MCI midpoint: mci midpoint instance: TBP>
end type mci_midpoint_instance_t

```

Output.

```

<MCI midpoint: mci midpoint instance: TBP>≡
    procedure :: write => mci_midpoint_instance_write

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_instance_write (object, unit)
        class(mci_midpoint_instance_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A,9(1x,F12.10))") "x =", object%x(:,1)
        write (u, "(1x,A,ES19.12)") "Integrand = ", object%integrand
        write (u, "(1x,A,ES19.12)") "Weight      = ", object%mci_weight
        if (object%max_known) then
            write (u, "(1x,A,ES19.12)") "Maximum    = ", object%max
            write (u, "(1x,A,ES19.12)") "Minimum    = ", object%min
            if (object%min /= object%min_abs) then
                write (u, "(1x,A,ES19.12)") "Max.(abs) = ", object%max_abs
                write (u, "(1x,A,ES19.12)") "Min.(abs) = ", object%min_abs
            end if
        end if
    end subroutine mci_midpoint_instance_write

```

The finalizer is empty.

```

<MCI midpoint: mci midpoint instance: TBP>+≡
    procedure :: final => mci_midpoint_instance_final

<MCI midpoint: tests>≡
    subroutine mci_midpoint_instance_final (object)
        class(mci_midpoint_instance_t), intent(inout) :: object
    end subroutine mci_midpoint_instance_final

```

Initializer.

```

<MCI midpoint: mci midpoint instance: TBP>+≡
    procedure :: init => mci_midpoint_instance_init

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_instance_init (mci_instance, mci)
        class(mci_midpoint_instance_t), intent(out) :: mci_instance
        class(mci_t), intent(in), target :: mci
        call mci_instance%base_init (mci)
        select type (mci)
        type is (mci_midpoint_t)
            mci_instance%mci => mci
            call mci_instance%get_max ()
            mci_instance%selected_channel = 1
        end select
    end subroutine mci_midpoint_instance_init

```

Copy the stored extrema of the integrand in the instance record.

```

(MCI midpoint: mci midpoint instance: TBP)+≡
  procedure :: get_max => mci_midpoint_instance_get_max

(MCI midpoint: procedures)+≡
  subroutine mci_midpoint_instance_get_max (instance)
    class(mci_midpoint_instance_t), intent(inout) :: instance
    associate (mci => instance%mci)
      if (mci%max_known) then
        instance%max_known = .true.
        instance%max = mci%max
        instance%min = mci%min
        instance%max_abs = mci%max_abs
        instance%min_abs = mci%min_abs
      end if
    end associate
  end subroutine mci_midpoint_instance_get_max

```

Reverse operations: recall the extrema, but only if they are wider than the extrema already stored in the configuration. Also recalculate the efficiency value.

```

(MCI midpoint: mci midpoint instance: TBP)+≡
  procedure :: set_max => mci_midpoint_instance_set_max

(MCI midpoint: procedures)+≡
  subroutine mci_midpoint_instance_set_max (instance)
    class(mci_midpoint_instance_t), intent(inout) :: instance
    associate (mci => instance%mci)
      if (instance%max_known) then
        if (mci%max_known) then
          mci%max = max (mci%max, instance%max)
          mci%min = min (mci%min, instance%min)
          mci%max_abs = max (mci%max_abs, instance%max_abs)
          mci%min_abs = min (mci%min_abs, instance%min_abs)
        else
          mci%max = instance%max
          mci%min = instance%min
          mci%max_abs = instance%max_abs
          mci%min_abs = instance%min_abs
          mci%max_known = .true.
        end if
        if (mci%max_abs /= 0) then
          if (mci%integral == mci%integral_pos) then
            mci%efficiency = mci%integral / mci%max_abs
            mci%efficiency_known = .true.
          else if (mci%n_calls /= 0) then
            mci%efficiency = &
              (mci%n_calls_pos * mci%integral_pos &
               - mci%n_calls_neg * mci%integral_neg) &
              / mci%n_calls / mci%max_abs
            mci%efficiency_known = .true.
          end if
        end if
      end if
    end associate
  end subroutine mci_midpoint_instance_set_max

```

```

        end associate
    end subroutine mci_midpoint_instance_set_max

```

The weight is the Jacobian of the mapping for the only channel.

```

<MCI midpoint: mci midpoint instance: TBP>+≡
    procedure :: compute_weight => mci_midpoint_instance_compute_weight

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_instance_compute_weight (mci, c)
        class(mci_midpoint_instance_t), intent(inout) :: mci
        integer, intent(in) :: c
        select case (c)
        case (1)
            mci%mci_weight = mci%f(1)
        case default
            call msg_fatal ("MCI midpoint integrator: only single channel supported")
        end select
    end subroutine mci_midpoint_instance_compute_weight

```

Record the integrand. Apply the Jacobian weight to get the absolute value. Divide by the channel maximum and by any overall factor to get the value relative to the maximum.

```

<MCI midpoint: mci midpoint instance: TBP>+≡
    procedure :: record_integrand => mci_midpoint_instance_record_integrand

<MCI midpoint: procedures>+≡
    subroutine mci_midpoint_instance_record_integrand (mci, integrand)
        class(mci_midpoint_instance_t), intent(inout) :: mci
        real(default), intent(in) :: integrand
        mci%integrand = integrand
        if (mci%max_known) then
            mci%max = max (mci%max, integrand)
            mci%min = min (mci%min, integrand)
            mci%max_abs = max (mci%max_abs, abs (integrand))
            mci%min_abs = min (mci%min_abs, abs (integrand))
        else
            mci%max = integrand
            mci%min = integrand
            mci%max_abs = abs (integrand)
            mci%min_abs = abs (integrand)
            mci%max_known = .true.
        end if
    end subroutine mci_midpoint_instance_record_integrand

```

### 13.2.3 Unit tests

```

<MCI midpoint: public>+≡
    public :: mci_midpoint_test

<MCI midpoint: tests>+≡
    subroutine mci_midpoint_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    end subroutine mci_midpoint_test

```



```

    <MCI midpoint: execute tests>
end subroutine mci_midpoint_test

```

## Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

This is the function  $f(x) = 3x^2$  with integral  $\int_0^1 f(x) dx = 1$  and maximum  $f(1) = 3$ . If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

Mimicking the behavior of a process object, we store the argument and result inside the sampler, so we can `fetch` results.

```

<MCI midpoint: test types>≡
type, extends (mci_sampler_t) :: test_sampler_1_t
  real(default), dimension(:), allocatable :: x
  real(default) :: val
contains
  <MCI midpoint: test sampler 1: TBP>
end type test_sampler_1_t

```

Output: There is nothing stored inside, so just print an informative line.

```

<MCI midpoint: test sampler 1: TBP>≡
  procedure :: write => test_sampler_1_write

<MCI midpoint: procedures>+≡
  subroutine test_sampler_1_write (object, unit)
    class(test_sampler_1_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
  end subroutine test_sampler_1_write

```

Evaluation: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

<MCI midpoint: test sampler 1: TBP>+≡
  procedure :: evaluate => test_sampler_1_evaluate

<MCI midpoint: procedures>+≡
  subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_1_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (allocated (sampler%x)) deallocate (sampler%x)
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    sampler%val = 3 * x_in(1) ** 2
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_1_evaluate

```

```
end subroutine test_sampler_1_evaluate
```

Rebuild: compute all but the function value.

```
<MCI midpoint: test sampler 1: TBP>+≡
  procedure :: rebuild => test_sampler_1_rebuild

<MCI midpoint: procedures>+≡
  subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_1_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (allocated (sampler%x)) deallocate (sampler%x)
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    sampler%val = val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_rebuild
```

Extract the results.

```
<MCI midpoint: test sampler 1: TBP>+≡
  procedure :: fetch => test_sampler_1_fetch

<MCI midpoint: procedures>+≡
  subroutine test_sampler_1_fetch (sampler, val, x, f)
    class(test_sampler_1_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_fetch
```

This is the function  $f(x) = 3x^2 + 2y$  with integral  $\int_0^1 f(x,y) dx dy = 2$  and maximum  $f(1) = 5$ .

```
<MCI midpoint: test types>+≡
  type, extends (mci_sampler_t) :: test_sampler_2_t
    real(default) :: val
    real(default), dimension(2) :: x
  contains
    <MCI midpoint: test sampler 2: TBP>
  end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

```
<MCI midpoint: test sampler 2: TBP>≡
  procedure :: write => test_sampler_2_write
```

```

<MCI midpoint: procedures>+≡
  subroutine test_sampler_2_write (object, unit)
    class(test_sampler_2_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2 + 2 y"
  end subroutine test_sampler_2_write

```

Evaluate: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

<MCI midpoint: test sampler 2: TBP>+≡
  procedure :: evaluate => test_sampler_2_evaluate

<MCI midpoint: procedures>+≡
  subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    sampler%x = x_in
    sampler%val = 3 * x_in(1) ** 2 + 2 * x_in(2)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_2_evaluate

```

Rebuild: compute all but the function value.

```

<MCI midpoint: test sampler 2: TBP>+≡
  procedure :: rebuild => test_sampler_2_rebuild

<MCI midpoint: procedures>+≡
  subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    sampler%x = x_in
    sampler%val = val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_2_rebuild

```

```

<MCI midpoint: test sampler 2: TBP>+≡
  procedure :: fetch => test_sampler_2_fetch

<MCI midpoint: procedures>+≡
  subroutine test_sampler_2_fetch (sampler, val, x, f)
    class(test_sampler_2_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f

```

```

    val = sampler%val
    x(:,1) = sampler%x
    f = 1
end subroutine test_sampler_2_fetch

```

This is the function  $f(x) = (1 - 3x^2)\theta(x - 1/2)$  with integral  $\int_0^1 f(x) dx = -3/8$ , minimum  $f(1) = -2$  and maximum  $f(1/2) = 1/4$ . If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

```

⟨MCI midpoint: test types⟩+≡
type, extends (mci_sampler_t) :: test_sampler_4_t
    real(default) :: val
    real(default), dimension(:), allocatable :: x
contains
    ⟨MCI midpoint: test sampler 4: TBP⟩
end type test_sampler_4_t

```

Output: There is nothing stored inside, so just print an informative line.

```

⟨MCI midpoint: test sampler 4: TBP⟩≡
procedure :: write => test_sampler_4_write

⟨MCI midpoint: procedures⟩+≡
subroutine test_sampler_4_write (object, unit)
    class(test_sampler_4_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Test sampler: f(x) = 1 - 3 x^2"
end subroutine test_sampler_4_write

```

Evaluation: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

⟨MCI midpoint: test sampler 4: TBP⟩+≡
procedure :: evaluate => test_sampler_4_evaluate

⟨MCI midpoint: procedures⟩+≡
subroutine test_sampler_4_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_4_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (x_in(1) >= .5_default) then
        sampler%val = 1 - 3 * x_in(1) ** 2
    else
        sampler%val = 0
    end if
    if (.not. allocated (sampler%x)) allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    call sampler%fetch (val, x, f)
end subroutine test_sampler_4_evaluate

```

Rebuild: compute all but the function value.

```

<MCI midpoint: test sampler 4: TBP>+≡
  procedure :: rebuild => test_sampler_4_rebuild

<MCI midpoint: procedures>+≡
  subroutine test_sampler_4_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_4_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    sampler%x = x_in
    sampler%val = val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_4_rebuild

```

```

<MCI midpoint: test sampler 4: TBP>+≡
  procedure :: fetch => test_sampler_4_fetch

```

```

<MCI midpoint: procedures>+≡
  subroutine test_sampler_4_fetch (sampler, val, x, f)
    class(test_sampler_4_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_4_fetch

```

## One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

```

<MCI midpoint: execute tests>≡
  call test (mci_midpoint_1, "mci_midpoint_1", &
    "one-dimensional integral", &
    u, results)

<MCI midpoint: tests>+≡
  subroutine mci_midpoint_1 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_midpoint_1"
    write (u, "(A)")  "* Purpose: integrate function in one dimension"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"

```

```

write (u, "(A)")

allocate (mci_midpoint_t :: mci)
call mci%set_dimensions (1, 1)

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_1_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for x = 0.8"
write (u, "(A)")

call mci_instance%evaluate (sampler, 1, [0.8_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for x = 0.7"
write (u, "(A)")

call mci_instance%evaluate (sampler, 1, [0.7_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for x = 0.9"
write (u, "(A)")

call mci_instance%evaluate (sampler, 1, [0.9_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_calls = 1000"
write (u, "(A)")

call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_1"

```

```
end subroutine mci_midpoint_1
```

## Two-dimensional integration

Construct an integrator and use it for a two-dimensional sampler.

```
(MCI midpoint: execute tests)+≡
  call test (mci_midpoint_2, "mci_midpoint_2", &
    "two-dimensional integral", &
    u, results)
(MCI midpoint: tests)+≡
  subroutine mci_midpoint_2 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_midpoint_2"
    write (u, "(A)")  "* Purpose: integrate function in two dimensions"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
    call mci%set_dimensions (2, 1)

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_2_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
    write (u, "(A)")

    call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_calls = 1000"
    write (u, "(A)")
```

```

call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_2"

end subroutine mci_midpoint_2

```

## Two-dimensional integration with flat dimension

Construct an integrator and use it for a two-dimensional sampler, where the function is constant in the second dimension.

```

<MCI midpoint: execute tests>+≡
call test (mci_midpoint_3, "mci_midpoint_3", &
  "two-dimensional integral with flat dimension", &
  u, results)

<MCI midpoint: tests>+≡
subroutine mci_midpoint_3 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  real(default) :: integrand

  write (u, "(A)")  "* Test output: mci_midpoint_3"
  write (u, "(A)")  "* Purpose: integrate function with one flat dimension"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  select type (mci)
  type is (mci_midpoint_t)
    call mci%set_dimensions (2, 1)
    call mci%declare_flat_dimensions ([2])
  end select

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"

```



```

write (u, "(A)")

allocate (test_sampler_1_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
write (u, "(A)")

call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_calls = 1000"
write (u, "(A)")

call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_3"

end subroutine mci_midpoint_3

```

### Integrand with sign flip

Construct an integrator and use it for a one-dimensional sampler.

```

<MCI midpoint: execute tests>+≡
  call test (mci_midpoint_4, "mci_midpoint_4", &
    "integrand with sign flip", &
    u, results)
<MCI midpoint: tests>+≡
  subroutine mci_midpoint_4 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_midpoint_4"
    write (u, "(A)")  "* Purpose: integrate function with sign flip &
      &in one dimension"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
    call mci%set_dimensions (1, 1)

```

```

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_4_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for x = 0.8"
write (u, "(A)")

call mci_instance%evaluate (sampler, 1, [0.8_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_calls = 1000"
write (u, "(A)")

call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_4"

end subroutine mci_midpoint_4

```

## Weighted events

Generate weighted events. Without rejection, we do not need to know maxima and minima, so we can start generating events immediately. We have two dimensions.

```

<MCI midpoint: execute tests>+≡
  call test (mci_midpoint_5, "mci_midpoint_5", &
    "weighted events", &
    u, results)

<MCI midpoint: tests>+≡
  subroutine mci_midpoint_5 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()

```

```

class(mci_sampler_t), allocatable :: sampler
class(rng_t), allocatable :: rng
class(mci_state_t), allocatable :: state

write (u, "(A)")  "* Test output: mci_midpoint_5"
write (u, "(A)")  "* Purpose: generate weighted events"
write (u, "(A)")

write (u, "(A)")  "* Initialize integrator"
write (u, "(A)")

allocate (mci_midpoint_t :: mci)
call mci%set_dimensions (2, 1)

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_2_t :: sampler)

write (u, "(A)")  "* Initialize random-number generator"
write (u, "(A)")

allocate (rng_test_t :: rng)
call rng%init ()
call mci%import_rng (rng)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Store data"
write (u, "(A)")

allocate (state)
call mci_instance%store (state)

```

```

call mci_instance%final ()
deallocate (mci_instance)

call state%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recall data and rebuild event"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)
call mci%rebuild_event (mci_instance, sampler, state)

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
deallocate (mci_instance)
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_5"

end subroutine mci_midpoint_5

```

## Unweighted events

Generate unweighted events. The integrand has a sign flip in it.

```

<MCI midpoint: execute tests>+≡
call test (mci_midpoint_6, "mci_midpoint_6", &
  "unweighted events", &
  u, results)

<MCI midpoint: tests>+≡
subroutine mci_midpoint_6 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_midpoint_6"
  write (u, "(A)")  "* Purpose: generate unweighted events"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  call mci%set_dimensions (1, 1)

```

```

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_4_t :: sampler)

write (u, "(A)")  "* Initialize random-number generator"
write (u, "(A)")

allocate (rng_test_t :: rng)
call rng%init ()
call mci%import_rng (rng)

write (u, "(A)")  "* Integrate (determine maximum of integrand)"
write (u, "(A)")
call mci%integrate (mci_instance, sampler, 1, 1000)

write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call mci%generate_unweighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
deallocate (mci_instance)
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_midpoint_6"

end subroutine mci_midpoint_6

```

### 13.3 VAMP interface

The standard method for integration is VAMP: the multi-channel version of the VEGAS algorithm. Each parameterization (channel) of the hypercube is binned in each dimension. The binning is equally equidistant, but an iteration of the integration procedure, the binning is updated for each dimension, according to the variance distribution of the integrand, summed over all other dimension. In the next iteration, the binning approximates (hopefully) follows the integrand

more closely, and the accuracy of the result is increased. Furthermore, the relative weight of the individual channels is also updated after an iteration.

The bin distribution is denoted as the grid for a channel, which we can write to file and reuse later.

In our implementation we specify the generic VAMP algorithm more tightly: the number of bins is equal for all dimensions, the initial weights are all equal. The user controls whether to update bins and/or weights after each iteration. The integration is organized in passes, each one consisting of several iterations with a common number of calls to the integrand. The first passes are intended as warmup, so the results are displayed but otherwise discarded. In the final pass, the integration estimates for the individual iterations are averaged for the final result.

```

<mci_vamp.f90>≡
  <File header>

  module mci_vamp

    use kinds !NODEP!
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use constants !NODEP!
    use unit_tests
    use md5

    use phs_base
    use rng_base
    use rng_tao
    use mci_base

    use vamp !NODEP!

    <Standard module head>

    <MCI vamp: public>

    <MCI vamp: types>

    <MCI vamp: interfaces>

    <MCI vamp: test types>

    contains

    <MCI vamp: procedures>

    <MCI vamp: tests>

  end module mci_vamp

```

### 13.3.1 Grid parameters

This is a transparent container. It holds the parameters that are stored in grid files, and are checked when grid files are read.

```
<MCI vamp: public>≡
    public :: grid_parameters_t

<MCI vamp: types>≡
    type :: grid_parameters_t
        integer :: threshold_calls = 0
        integer :: min_calls_per_channel = 10
        integer :: min_calls_per_bin = 10
        integer :: min_bins = 3
        integer :: max_bins = 20
        logical :: stratified = .true.
        logical :: use_vamp_equivalences = .true.
        real(default) :: channel_weights_power = 0.25_default
    contains
        <MCI vamp: grid parameters: TBP>
    end type grid_parameters_t
```

I/O:

```
<MCI vamp: grid parameters: TBP>≡
    procedure :: write => grid_parameters_write

<MCI vamp: procedures>≡
    subroutine grid_parameters_write (object, unit)
        class(grid_parameters_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(3x,A,I0)") "threshold_calls      = ", &
            object%threshold_calls
        write (u, "(3x,A,I0)") "min_calls_per_channel = ", &
            object%min_calls_per_channel
        write (u, "(3x,A,I0)") "min_calls_per_bin   = ", &
            object%min_calls_per_bin
        write (u, "(3x,A,I0)") "min_bins           = ", &
            object%min_bins
        write (u, "(3x,A,I0)") "max_bins           = ", &
            object%max_bins
        write (u, "(3x,A,L1)") "stratified         = ", &
            object%stratified
        write (u, "(3x,A,L1)") "use_vamp_equivalences = ", &
            object%use_vamp_equivalences
        write (u, "(3x,A,F10.7)") "channel_weights_power = ", &
            object%channel_weights_power
    end subroutine grid_parameters_write

<XXX MCI vamp: procedures>≡
    subroutine grid_parameters_read (grid_par, unit)
        type(grid_parameters_t), intent(out) :: grid_par
        integer, intent(in) :: unit
        character(30) :: dummy
```

```

character :: equals
read (unit, *) dummy, equals, grid_par%threshold_calls
read (unit, *) dummy, equals, grid_par%min_calls_per_channel
read (unit, *) dummy, equals, grid_par%min_calls_per_bin
read (unit, *) dummy, equals, grid_par%min_bins
read (unit, *) dummy, equals, grid_par%max_bins
read (unit, *) dummy, equals, grid_par%stratified
read (unit, *) dummy, equals, grid_par%use_vamp_equivalences
read (unit, *) dummy, equals, grid_par%channel_weights_power
end subroutine grid_parameters_read

```

### 13.3.2 Integration pass

We store the parameters for each integration pass in a linked list.

```

⟨MCI vamp: types⟩+=
  type :: pass_t
    integer :: i_pass = 0
    integer :: i_first_it = 0
    integer :: n_it = 0
    integer :: n_calls = 0
    integer :: n_bins = 0
    logical :: adapt_grids = .false.
    logical :: adapt_weights = .false.
    logical :: integral_defined = .false.
    integer, dimension(:), allocatable :: calls
    real(default), dimension(:), allocatable :: integral
    real(default), dimension(:), allocatable :: error
    !   type(vamp_history), dimension(:), allocatable :: history
    type(pass_t), pointer :: next => null ()
  contains
    ⟨MCI vamp: pass: TBP⟩
  end type pass_t

```

Output. Note that the precision of the numerical values should match the precision for comparing output from file with data.

```

⟨MCI vamp: pass: TBP⟩=
  procedure :: write => pass_write

⟨MCI vamp: procedures⟩+=
  subroutine pass_write (object, unit)
    class(pass_t), intent(in) :: object
    integer, intent(in) :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(3x,A,I0)") "n_it           = ", object%n_it
    write (u, "(3x,A,I0)") "n_calls        = ", object%n_calls
    write (u, "(3x,A,I0)") "n_bins         = ", object%n_bins
    write (u, "(3x,A,L1)") "adapt grids    = ", object%adapt_grids
    write (u, "(3x,A,L1)") "adapt weights = ", object%adapt_weights
    if (object%integral_defined) then
      write (u, "(3x,A)") "Results: [it, calls, integral, error]"
      do i = 1, object%n_it

```



```

        write (u, "(5x,I0,1x,I0,2(1x,ES17.10))") &
            i, object%calls(i), object%integral(i), object%error(i)
    end do
else
    write (u, "(3x,A)") "Results: [undefined]"
end if
end subroutine pass_write

```

Read and reconstruct the pass.

```

<MCI vamp: pass: TBP>+≡
    procedure :: read => pass_read

<MCI vamp: procedures>+≡
    subroutine pass_read (object, u, n_pass, n_it)
        class(pass_t), intent(out) :: object
        integer, intent(in) :: u, n_pass, n_it
        integer :: i, j
        character(80) :: buffer
        object%i_pass = n_pass + 1
        object%i_first_it = n_it + 1
        call read_ival (u, object%n_it)
        call read_ival (u, object%n_calls)
        call read_ival (u, object%n_bins)
        call read_lval (u, object%adapt_grids)
        call read_lval (u, object%adapt_weights)
        allocate (object%calls (object%n_it), source = 0)
        allocate (object%integral (object%n_it), source = 0._default)
        allocate (object%error (object%n_it), source = 0._default)
        read (u, "(A)") buffer
        select case (trim (adjustl (buffer)))
        case ("Results: [it, calls, integral, error]")
            do i = 1, object%n_it
                read (u, *) &
                    j, object%calls(i), object%integral(i), object%error(i)
            end do
            object%integral_defined = .true.
        case ("Results: [undefined]")
            object%integral_defined = .false.
        case default
            call msg_fatal ("Reading integration pass: corrupted file")
        end select
    end subroutine pass_read

```

Given a number of calls and iterations, compute remaining data.

```

<MCI vamp: pass: TBP>+≡
    procedure :: configure => pass_configure

<MCI vamp: procedures>+≡
    subroutine pass_configure (pass, n_it, n_calls, min_calls, min_bins, max_bins)
        class(pass_t), intent(inout) :: pass
        integer, intent(in) :: n_it, n_calls
        integer, intent(in) :: min_calls, min_bins, max_bins
        pass%n_it = n_it
        pass%n_calls = n_calls

```

```

if (min_calls /= 0) then
  pass%n_bins = max (min_bins, &
    min (n_calls / min_calls, max_bins))
else
  pass%n_bins = max_bins
end if
allocate (pass%calls (n_it), source = 0)
allocate (pass%integral (n_it), source = 0._default)
allocate (pass%error (n_it), source = 0._default)
end subroutine pass_configure

```

Given two pass objects, compare them. All parameters must match. Where integrations are done in both (number of calls nonzero), the results must be equal (up to numerical noise).

```

<MCI vamp: interfaces>≡
  interface operator (.matches.)
    module procedure pass_matches
  end interface operator (.matches.)

<MCI vamp: procedures>+≡
  function pass_matches (pass, ref) result (ok)
    type(pass_t), intent(in) :: pass, ref
    logical :: ok
    ok = .true.
    if (ok) ok = pass%i_pass == ref%i_pass
    if (ok) ok = pass%i_first_it == ref%i_first_it
    if (ok) ok = pass%n_it == ref%n_it
    if (ok) ok = pass%n_calls == ref%n_calls
    if (ok) ok = pass%n_bins == ref%n_bins
    if (ok) ok = pass%adapt_grids .eqv. ref%adapt_grids
    if (ok) ok = pass%adapt_weights .eqv. ref%adapt_weights
    if (ok) ok = pass%integral_defined .eqv. ref%integral_defined
    if (pass%integral_defined) then
      if (ok) ok = all (pass%calls == ref%calls)
      if (ok) ok = all (pass%integral .matches. ref%integral)
      if (ok) ok = all (pass%error .matches. ref%error)
    end if
  end function pass_matches

```

Update a pass object, given a reference. The parameters must match, except for the `n_it` entry. The number of complete iterations must be less or equal to the reference, and the number of complete iterations in the reference must be no larger than `n_it`. Where results are present in both passes, they must match. Where results are present in the reference only, the pass is updated accordingly.

```

<MCI vamp: pass: TBP>+≡
  procedure :: update => pass_update

<MCI vamp: procedures>+≡
  subroutine pass_update (pass, ref, ok)
    class(pass_t), intent(inout) :: pass
    type(pass_t), intent(in) :: ref
    logical, intent(out) :: ok
    integer :: n, n_ref

```

```

ok = .true.
if (ok) ok = pass%i_pass == ref%i_pass
if (ok) ok = pass%i_first_it == ref%i_first_it
if (ok) ok = pass%n_calls == ref%n_calls
if (ok) ok = pass%n_bins == ref%n_bins
if (ok) ok = pass%adapt_grids .eqv. ref%adapt_grids
if (ok) ok = pass%adapt_weights .eqv. ref%adapt_weights
if (ok) then
  if (ref%integral_defined) then
    if (.not. allocated (pass%calls)) then
      allocate (pass%calls (pass%n_it), source = 0)
      allocate (pass%integral (pass%n_it), source = 0._default)
      allocate (pass%error (pass%n_it), source = 0._default)
    end if
    n = count (pass%calls /= 0)
    n_ref = count (ref%calls /= 0)
    ok = n <= n_ref .and. n_ref <= pass%n_it
    if (ok) ok = all (pass%calls(:n) == ref%calls(:n))
    if (ok) ok = all (pass%integral(:n) .matches. ref%integral(:n))
    if (ok) ok = all (pass%error(:n) .matches. ref%error(:n))
    if (ok) then
      pass%calls(n+1:n_ref) = ref%calls(n+1:n_ref)
      pass%integral(n+1:n_ref) = ref%integral(n+1:n_ref)
      pass%error(n+1:n_ref) = ref%error(n+1:n_ref)
      pass%integral_defined = any (pass%calls /= 0)
    end if
  end if
end if
end subroutine pass_update

```

Match two real numbers: they are equal up to a tolerance, which is  $10^{-8}$ , matching the number of digits that are output by `pass_write`. In particular, if one number is exactly zero, the other one must also be zero.

```

(MCI vamp: interfaces)+≡
  interface operator (.matches.)
    module procedure real_matches
  end interface operator (.matches.)

(MCI vamp: procedures)+≡
  elemental function real_matches (x, y) result (ok)
    real(default), intent(in) :: x, y
    logical :: ok
    real(default), parameter :: tolerance = 1.e-8_default
    ok = abs (x - y) <= tolerance * max (abs (x), abs (y))
  end function real_matches

```

Return the index of the most recent complete integration. If there is none, return zero.

```

(MCI vamp: pass: TBP)+≡
  procedure :: get_integration_index => pass_get_integration_index

(MCI vamp: procedures)+≡
  function pass_get_integration_index (pass) result (n)
    class (pass_t), intent(in) :: pass

```

```

integer :: n
integer :: i
n = 0
if (allocated (pass%calls)) then
  do i = 1, pass%n_it
    if (pass%calls(i) == 0) exit
    n = i
  end do
end if
end function pass_get_integration_index

```

Return the most recent integral and error, if available.

*(MCI vamp: pass: TBP)+≡*

```

procedure :: get_calls => pass_get_calls
procedure :: get_integral => pass_get_integral
procedure :: get_error => pass_get_error

```

*(MCI vamp: procedures)+≡*

```

function pass_get_calls (pass) result (calls)
  class(pass_t), intent(in) :: pass
  integer :: calls
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
    calls = pass%calls(n)
  else
    calls = 0
  end if
end function pass_get_calls

function pass_get_integral (pass) result (integral)
  class(pass_t), intent(in) :: pass
  real(default) :: integral
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
    integral = pass%integral(n)
  else
    integral = 0
  end if
end function pass_get_integral

function pass_get_error (pass) result (error)
  class(pass_t), intent(in) :: pass
  real(default) :: error
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
    error = pass%error(n)
  else
    error = 0
  end if
end function pass_get_error

```

### 13.3.3 Integrator

```

<MCI vamp: public>+≡
    public :: mci_vamp_t

<MCI vamp: types>+≡
    type, extends (mci_t) :: mci_vamp_t
        logical, dimension(:), allocatable :: dim_is_flat
        type(grid_parameters_t) :: grid_par
        integer :: min_calls = 0
        type(pass_t), pointer :: first_pass => null ()
        type(pass_t), pointer :: current_pass => null ()
        type(vamp_equivalences_t) :: equivalences
        logical :: rebuild = .true.
        logical :: grid_filename_set = .false.
        type(string_t) :: grid_filename
        character(32) :: md5sum_adapted = ""
    contains
        <MCI vamp: mci vamp: TBP>
    end type mci_vamp_t

```

Reset: delete integration-pass entries.

```

<MCI vamp: mci vamp: TBP>≡
    procedure :: reset => mci_vamp_reset

<MCI vamp: procedures>+≡
    subroutine mci_vamp_reset (object)
        class(mci_vamp_t), intent(inout) :: object
        type(pass_t), pointer :: current_pass
        do while (associated (object%first_pass))
            current_pass => object%first_pass
            object%first_pass => current_pass%next
            deallocate (current_pass)
        end do
        object%current_pass => null ()
    end subroutine mci_vamp_reset

```

Finalizer: reset and finalize the equivalences list.

```

<MCI vamp: mci vamp: TBP>+≡
    procedure :: final => mci_vamp_final

<MCI vamp: procedures>+≡
    subroutine mci_vamp_final (object)
        class(mci_vamp_t), intent(inout) :: object
        type(pass_t), pointer :: current_pass
        call object%reset ()
        call vamp_equivalences_final (object%equivalences)
    end subroutine mci_vamp_final

```

Output. Do not output the grids themselves, this may result in tons of data.

```

<MCI vamp: mci vamp: TBP>+≡
    procedure :: write => mci_vamp_write

```

```

<MCI vamp: procedures>+=
subroutine mci_vamp_write (object, unit)
  class(mci_vamp_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(pass_t), pointer :: current_pass
  integer :: u, i
  u = output_unit (unit)
  write (u, "(1x,A)") "VAMP integrator:"
  call object%base_write (u)
  if (allocated (object%dim_is_flat)) then
    write (u, "(3x,A,999(1x,I0))") "Flat dimensions      =", &
      pack ([i, i = 1, object%n_dim], object%dim_is_flat)
  end if
  write (u, "(1x,A)") "Grid parameters:"
  call object%grid_par%write (u)
  write (u, "(3x,A,I0)") "min_calls              = ", object%min_calls
  if (object%grid_par%use_vamp_equivalences) then
    call vamp_equivalences_write (object%equivalences, u)
  end if
  current_pass => object%first_pass
  do while (associated (current_pass))
    write (u, "(1x,A,I0,A)") "Integration pass:"
    call current_pass%write (u)
    current_pass => current_pass%next
  end do
  if (object%md5sum_adapted /= "") then
    write (u, "(1x,A,A,A)") "MD5 sum (including results) = '", &
      object%md5sum_adapted, "'"
  end if
end subroutine mci_vamp_write

```

Compute the MD5 sum, including the configuration MD5 sum and the printout, which incorporates the current results.

```

<MCI vamp: mci vamp: TBP>+=
  procedure :: compute_md5sum => mci_vamp_compute_md5sum

<MCI vamp: procedures>+=
subroutine mci_vamp_compute_md5sum (mci)
  class(mci_vamp_t), intent(inout) :: mci
  integer :: u
  mci%md5sum_adapted = ""
  u = free_unit ()
  open (u, status = "scratch", action = "readwrite")
  write (u, "(A)") mci%md5sum
  call mci%write (u)
  rewind (u)
  mci%md5sum_adapted = md5sum (u)
  close (u)
end subroutine mci_vamp_compute_md5sum

```

Return the MD5 sum: If available, return the adapted one.

```

<MCI vamp: mci vamp: TBP>+=
  procedure :: get_md5sum => mci_vamp_get_md5sum

```

```

<MCI vamp: procedures>+=
function mci_vamp_get_md5sum (mci) result (md5sum)
  class(mci_vamp_t), intent(in) :: mci
  character(32) :: md5sum
  if (mci%md5sum_adapted /= "") then
    md5sum = mci%md5sum_adapted
  else
    md5sum = mci%md5sum
  end if
end function mci_vamp_get_md5sum

```

Startup message: short version.

```

<MCI vamp: mci vamp: TBP>+=
procedure :: startup_message => mci_vamp_startup_message

<MCI vamp: procedures>+=
subroutine mci_vamp_startup_message (mci, unit)
  class(mci_vamp_t), intent(in) :: mci
  integer, intent(in), optional :: unit
  call mci%base_startup_message (unit)
  write (msg_buffer, "(A,2(1x,I0,1x,A))") &
    "Integrator: VAMP"
  call msg_message (unit = unit)
end subroutine mci_vamp_startup_message

```

Set the grid parameters.

```

<MCI vamp: mci vamp: TBP>+=
procedure :: set_grid_parameters => mci_vamp_set_grid_parameters

<MCI vamp: procedures>+=
subroutine mci_vamp_set_grid_parameters (mci, grid_par)
  class(mci_vamp_t), intent(inout) :: mci
  type(grid_parameters_t), intent(in) :: grid_par
  mci%grid_par = grid_par
  mci%min_calls = grid_par%min_calls_per_bin * mci%n_channel
end subroutine mci_vamp_set_grid_parameters

```

Set the rebuild flag.

```

<MCI vamp: mci vamp: TBP>+=
procedure :: set_rebuild_flag => mci_vamp_set_rebuild_flag

<MCI vamp: procedures>+=
subroutine mci_vamp_set_rebuild_flag (mci, rebuild)
  class(mci_vamp_t), intent(inout) :: mci
  logical, intent(in) :: rebuild
  mci%rebuild = rebuild
end subroutine mci_vamp_set_rebuild_flag

```

Set the filename.

```

<MCI vamp: mci vamp: TBP>+=
procedure :: set_grid_filename => mci_vamp_set_grid_filename

```

```

<MCI vamp: procedures>+=
  subroutine mci_vamp_set_grid_filename (mci, name, run_id)
    class(mci_vamp_t), intent(inout) :: mci
    type(string_t), intent(in) :: name
    type(string_t), intent(in), optional :: run_id
    if (present (run_id)) then
      mci%grid_filename = name // "." // run_id // ".vg"
    else
      mci%grid_filename = name // ".vg"
    end if
    mci%grid_filename_set = .true.
  end subroutine mci_vamp_set_grid_filename

```

Declare particular dimensions as flat.

```

<MCI vamp: mci vamp: TBP>+=
  procedure :: declare_flat_dimensions => mci_vamp_declare_flat_dimensions

<MCI vamp: procedures>+=
  subroutine mci_vamp_declare_flat_dimensions (mci, dim_flat)
    class(mci_vamp_t), intent(inout) :: mci
    integer, dimension(:), intent(in) :: dim_flat
    integer :: d
    allocate (mci%dim_is_flat (mci%n_dim), source = .false.)
    do d = 1, size (dim_flat)
      mci%dim_is_flat(dim_flat(d)) = .true.
    end do
  end subroutine mci_vamp_declare_flat_dimensions

```

Declare equivalences. We have an array of channel equivalences, provided by the phase-space module. Here, we translate this into the `vamp_equivalences` array.

```

<MCI vamp: mci vamp: TBP>+=
  procedure :: declare_equivalences => mci_vamp_declare_equivalences

<MCI vamp: procedures>+=
  subroutine mci_vamp_declare_equivalences (mci, channel, dim_offset)
    class(mci_vamp_t), intent(inout) :: mci
    type(phs_channel_t), dimension(:), intent(in) :: channel
    integer, intent(in) :: dim_offset
    integer, dimension(:), allocatable :: perm, mode
    integer :: n_channels, n_dim, n_equivalences
    integer :: c, i, j, left, right
    n_channels = mci%n_channel
    n_dim = mci%n_dim
    n_equivalences = 0
    do c = 1, n_channels
      n_equivalences = n_equivalences + size (channel(c)%eq)
    end do
    call vamp_equivalences_init (mci%equivalences, &
      n_equivalences, n_channels, n_dim)
    allocate (perm (n_dim))
    allocate (mode (n_dim))
    perm(1:dim_offset) = [(i, i = 1, dim_offset)]
    mode(1:dim_offset) = VEQ_IDENTITY

```



```

c = 1
j = 0
do i = 1, n_equivalences
  if (j < size (channel(c)%eq)) then
    j = j + 1
  else
    c = c + 1
    j = 1
  end if
  associate (eq => channel(c)%eq(j))
    left = c
    right = eq%c
    perm(dim_offset+1:) = eq%perm + dim_offset
    mode(dim_offset+1:) = eq%mode
    call vamp_equivalence_set (mci%equivalences, &
      i, left, right, perm, mode)
  end associate
end do
call vamp_equivalences_complete (mci%equivalences)
end subroutine mci_vamp_declare_equivalences

```

Allocate instance with matching type.

```

⟨MCI vamp: mci vamp: TBP⟩+≡
  procedure :: allocate_instance => mci_vamp_allocate_instance

⟨MCI vamp: procedures⟩+≡
  subroutine mci_vamp_allocate_instance (mci, mci_instance)
    class(mci_vamp_t), intent(in) :: mci
    class(mci_instance_t), intent(out), pointer :: mci_instance
    allocate (mci_vamp_instance_t :: mci_instance)
  end subroutine mci_vamp_allocate_instance

```

Allocate a new integration pass. We can preset everything that does not depend on the number of iterations and calls. This is postponed to the `integrate` method.

```

⟨MCI vamp: mci vamp: TBP⟩+≡
  procedure :: add_pass => mci_vamp_add_pass

⟨MCI vamp: procedures⟩+≡
  subroutine mci_vamp_add_pass (mci, adapt_grids, adapt_weights)
    class(mci_vamp_t), intent(inout) :: mci
    logical, intent(in), optional :: adapt_grids, adapt_weights
    integer :: i_pass, i_it
    type(pass_t), pointer :: new
    allocate (new)
    if (associated (mci%current_pass)) then
      i_pass = mci%current_pass%i_pass + 1
      i_it = mci%current_pass%i_first_it + mci%current_pass%n_it
      mci%current_pass%next => new
    else
      i_pass = 1
      i_it = 1
      mci%first_pass => new
    end if
  end subroutine

```

```

mci%current_pass => new
new%i_pass = i_pass
new%i_first_it = i_it
if (present (adapt_grids)) then
    new%adapt_grids = adapt_grids
else
    new%adapt_grids = .false.
end if
if (present (adapt_weights)) then
    new%adapt_weights = adapt_weights
else
    new%adapt_weights = .false.
end if
end subroutine mci_vamp_add_pass

```

Update the list of integration passes. All passes except for the last one must match exactly. For the last one, integration results are updated. The reference output may contain extra passes, these are ignored.

*(MCI vamp: mci vamp: TBP)+≡*

```

procedure :: update_from_ref => mci_vamp_update_from_ref

```

*(MCI vamp: procedures)+≡*

```

subroutine mci_vamp_update_from_ref (mci, mci_ref, success)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_t), intent(in) :: mci_ref
    logical, intent(out) :: success
    type(pass_t), pointer :: current_pass, ref_pass
    select type (mci_ref)
    type is (mci_vamp_t)
        current_pass => mci%first_pass
        ref_pass => mci_ref%first_pass
        success = .true.
        do while (success .and. associated (current_pass))
            if (associated (ref_pass)) then
                if (associated (current_pass%next)) then
                    success = current_pass .matches. ref_pass
                else
                    call current_pass%update (ref_pass, success)
                    if (current_pass%integral_defined) then
                        mci%integral = current_pass%get_integral ()
                        mci%error = current_pass%get_error ()
                        mci%integral_known = .true.
                        mci%error_known = .true.
                    end if
                end if
                current_pass => current_pass%next
                ref_pass => ref_pass%next
            else
                success = .false.
            end if
        end do
    end select
end subroutine mci_vamp_update_from_ref

```

Update the MCI record (i.e., the integration passes) by reading from input stream. The stream should contain a `write` output from a previous run. We first check the MD5 sum of the configuration parameters. If that matches, we proceed directly to the stored integration passes. If successful, we may continue to read the file; the position will be after a blank line that must follow the MCI record.

```

(MCI vamp: mci vamp: TBP)+≡
  procedure :: update => mci_vamp_update

(MCI vamp: procedures)+≡
  subroutine mci_vamp_update (mci, u, success)
    class(mci_vamp_t), intent(inout) :: mci
    integer, intent(in) :: u
    logical, intent(out) :: success
    character(80) :: buffer
    character(32) :: md5sum_file
    type(mci_vamp_t) :: mci_file
    integer :: n_pass, n_it
    call read_sval (u, md5sum_file)
    success = md5sum_file == mci%md5sum
    if (success) then
      read (u, *)
      read (u, "(A)") buffer
      if (trim (adjustl (buffer))) == "VAMP integrator:") then
        n_pass = 0
        n_it = 0
        do
          read (u, "(A)") buffer
          select case (trim (adjustl (buffer)))
            case ("")
              exit
            case ("Integration pass:")
              call mci_file%add_pass ()
              call mci_file%current_pass%read (u, n_pass, n_it)
              n_pass = n_pass + 1
              n_it = n_it + mci_file%current_pass%n_it
            end select
          end do
          call mci%update_from_ref (mci_file, success)
          call mci_file%final ()
        else
          call msg_fatal ("vamp: reading grid file: corrupted data")
        end if
      end if
    end subroutine mci_vamp_update

```

Read / write grids from / to file.

```

(MCI vamp: mci vamp: TBP)+≡
  procedure :: write_grids => mci_vamp_write_grids
  procedure :: read_grids => mci_vamp_read_grids

(MCI vamp: procedures)+≡
  subroutine mci_vamp_write_grids (mci, instance)
    class(mci_vamp_t), intent(in) :: mci

```

```

class(mci_instance_t), intent(inout) :: instance
integer :: u
select type (instance)
type is (mci_vamp_instance_t)
  if (mci%grid_filename_set) then
    if (instance%grids_defined) then
      u = free_unit ()
      open (u, file = char (mci%grid_filename), &
        action = "write", status = "replace")
      write (u, "(1x,A,A,A)") "MD5sum = ', mci%md5sum, "'
      write (u, *)
      call mci%write (u)
      write (u, *)
      write (u, "(1x,A)") "VAMP grids:"
      call vamp_write_grids (instance%grids, u)
      close (u)
    else
      call msg_bug ("vamp: write grids: grids undefined")
    end if
  else
    call msg_bug ("vamp: write grids: filename undefined")
  end if
end select
end subroutine mci_vamp_write_grids

subroutine mci_vamp_read_grids (mci, instance, success)
class(mci_vamp_t), intent(inout) :: mci
class(mci_instance_t), intent(inout) :: instance
logical, intent(out) :: success
logical :: exist
integer :: u, p
character(80) :: buffer
select type (instance)
type is (mci_vamp_instance_t)
  success = .false.
  if (mci%grid_filename_set) then
    if (.not. instance%grids_defined) then
      inquire (file = char (mci%grid_filename), exist = exist)
      if (exist) then
        u = free_unit ()
        open (u, file = char (mci%grid_filename), &
          action = "read", status = "old")
        call mci%update (u, success)
        if (success) then
          read (u, "(A)") buffer
          if (trim (adjustl (buffer)) == "VAMP grids:") then
            call vamp_read_grids (instance%grids, u)
          else
            call msg_fatal ("vamp: reading grid file: &
              &corrupted grid data")
          end if
        else
          write (msg_buffer, "(A,A,A)") &
            "vamp: parameter mismatch, discarding grid file '", &

```

```

        char (mci%grid_filename), ""
        call msg_message ()
    end if
    close (u)
    instance%grids_defined = success
end if
else
    call msg_bug ("vamp: read grids: grids already defined")
end if
else
    call msg_bug ("vamp: read grids: filename undefined")
end if
end select
end subroutine mci_vamp_read_grids

```

Auxiliary: Read real, integer, string value. We search for an equals sign, the value must follow.

*(MCI vamp: procedures)+≡*

```

subroutine read_rval (u, rval)
    integer, intent(in) :: u
    real(default), intent(out) :: rval
    character(80) :: buffer
    read (u, "(A)") buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *) rval
end subroutine read_rval

subroutine read_ival (u, ival)
    integer, intent(in) :: u
    integer, intent(out) :: ival
    character(80) :: buffer
    read (u, "(A)") buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *) ival
end subroutine read_ival

subroutine read_sval (u, sval)
    integer, intent(in) :: u
    character(*), intent(out) :: sval
    character(80) :: buffer
    read (u, "(A)") buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *) sval
end subroutine read_sval

subroutine read_lval (u, lval)
    integer, intent(in) :: u
    logical, intent(out) :: lval
    character(80) :: buffer
    read (u, "(A)") buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *) lval
end subroutine read_lval

```

Integrate. Perform a new integration pass (possibly reusing previous results), which may consist of several iterations.

Note: we record the integral once per iteration. The integral stored in the `mci` record itself is the last integral of the current iteration, no averaging done. The `results` record may average results.

Note: recording the efficiency is not supported yet.

*(MCI vamp: mci vamp: TBP)+≡*

```
procedure :: integrate => mci_vamp_integrate
```

*(MCI vamp: procedures)+≡*

```
subroutine mci_vamp_integrate (mci, instance, sampler, n_it, n_calls, results)
  class(mci_vamp_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  class(mci_results_t), intent(inout), optional :: results
  integer :: it
  logical :: reshape, from_file, success
  select type (instance)
  type is (mci_vamp_instance_t)
    if (associated (mci%current_pass)) then
      mci%current_pass%integral_defined = .false.
      call mci%current_pass%configure (n_it, n_calls, &
        mci%min_calls, &
        mci%grid_par%min_bins, mci%grid_par%max_bins)
      instance%pass_complete = .false.
      instance%it_complete = .false.
      call instance%new_pass (reshape)
      if (.not. instance%grids_defined) then
        if (mci%grid_filename_set .and. .not. mci%rebuild) then
          call mci%read_grids (instance, success)
        else
          success = .false.
        end if
      if (success) then
        call msg_message ("vamp: " &
          // "using grids and results from file '" &
          // char (mci%grid_filename) // "'")
      else
        call instance%create_grids ()
      end if
    end if
  do it = 1, instance%n_it
    from_file = it < mci%current_pass%get_integration_index ()
    if (.not. from_file) then
      instance%it_complete = .false.
      call instance%adapt_grids ()
      call instance%adapt_weights ()
      call instance%discard_integrals (reshape)
      if (mci%grid_par%use_vamp_equivalences) then
        call instance%sample_grids (mci%rng, sampler, mci%equivalences)
      else
```

```

        call instance%sample_grids (mci%rng, sampler)
    end if
    instance%it_complete = .true.
    if (instance%integral /= 0) then
        mci%current_pass%calls(it) = instance%calls
        mci%current_pass%integral(it) = instance%integral
        if (abs (instance%error / instance%integral) &
            > epsilon (1._default)) then
            mci%current_pass%error(it) = instance%error
        end if
    end if
    mci%current_pass%integral_defined = .true.
end if
if (present (results)) then
    call results%record (1, &
        n_calls      = mci%current_pass%calls(it), &
        integral     = mci%current_pass%integral(it), &
        error        = mci%current_pass%error(it), &
        efficiency    = 0._default)
end if
if (.not. from_file .and. mci%grid_filename_set) then
    call mci%write_grids (instance)
end if
call instance%allow_adaptation ()
reshape = .false.
end do
instance%pass_complete = .true.
mci%integral = mci%current_pass%get_integral()
mci%error = mci%current_pass%get_error()
! mci%efficiency not yet calculated
mci%integral_known = .true.
mci%error_known = .true.
call mci%compute_md5sum ()
else
    call msg_bug ("MCI integrate: current_pass object not allocated")
end if
end select
end subroutine mci_vamp_integrate

```

Prepare an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

The pass-specific data of the previous integration pass are retained, but we reset the number of iterations and calls to zero. The latter now counts the number of events (calls to the sampling function, actually).

```

<MCI vamp: mci vamp: TBP>+≡
    procedure :: init_simulation => mci_vamp_init_simulation

<MCI vamp: procedures>+≡
    subroutine mci_vamp_init_simulation (mci, instance)
        class(mci_vamp_t), intent(inout) :: mci
        class(mci_instance_t), intent(inout) :: instance
        logical :: success
        select type (instance)
            type is (mci_vamp_instance_t)

```

```

allocate (instance%vamp_x (instance%nci%n_dim))
instance%it = 0
instance%calls = 0
instance%generating_events = .true.
if (.not. instance%grids_defined) then
  if (mci%grid_filename_set) then
    call mci%read_grids (instance, success)
    call mci%compute_md5sum ()
    if (success) then
      call msg_message ("Simulate: " &
        // "using integration grids from file '" &
        // char (mci%grid_filename) // "'")
    else
      call msg_fatal ("Simulate: " &
        // "reading integration grids from file '" &
        // char (mci%grid_filename) // "' failed")
    end if
  else
    call msg_bug ("vamp: simulation: no grids, no grid filename")
  end if
end if
end select
end subroutine mci_vamp_init_simulation

```

Finalize an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

```

<MCI vamp: mci vamp: TBP>+≡
  procedure :: final_simulation => mci_vamp_final_simulation

<MCI vamp: procedures>+≡
  subroutine mci_vamp_final_simulation (mci, instance)
    class(mci_vamp_t), intent(in) :: mci
    class(mci_instance_t), intent(inout) :: instance
    select type (instance)
      type is (mci_vamp_instance_t)
        deallocate (instance%vamp_x)
      end select
    end select
  end subroutine mci_vamp_final_simulation

```

Generate weighted event. Note that the event weight (`vamp_weight`) is not just the MCI weight. `vamp_next_event` selects a channel based on the channel weights multiplied by the (previously recorded) maximum integrand value of the channel. The MCI weight is renormalized accordingly, to cancel this effect on the result.

```

<MCI vamp: mci vamp: TBP>+≡
  procedure :: generate_weighted_event => mci_vamp_generate_weighted_event

<MCI vamp: procedures>+≡
  subroutine mci_vamp_generate_weighted_event (mci, instance, sampler)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    class(vamp_data_t), allocatable :: data

```



```

select type (instance)
type is (mci_vamp_instance_t)
  instance%vamp_weight_set = .false.
  allocate (mci_workspace_t :: data)
  select type (data)
  type is (mci_workspace_t)
    data%sampler => sampler
    data%instance => instance
  end select
  select type (rng => mci%rng)
  type is (rng_tao_t)
    if (instance%grids_defined) then
      call vamp_next_event ( &
        instance%vamp_x, &
        rng%state, &
        instance%grids, &
        vamp_sampling_function, &
        data, &
        phi = phi_trivial, &
        weight = instance%vamp_weight)
      instance%vamp_weight_set = .true.
    else
      call msg_bug ("vamp: generate event: grids undefined")
    end if
  class default
    call msg_fatal ("VAMP event generation: &
      &random-number generator must be TAO")
  end select
end select
end subroutine mci_vamp_generate_weighted_event

```

Generate unweighted event.

*(MCI vamp: mci vamp: TBP)+≡*

```

procedure :: generate_unweighted_event => &
  mci_vamp_generate_unweighted_event

```

*(MCI vamp: procedures)+≡*

```

subroutine mci_vamp_generate_unweighted_event (mci, instance, sampler)
  class(mci_vamp_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  class(vamp_data_t), allocatable :: data
  select type (instance)
  type is (mci_vamp_instance_t)
    instance%vamp_weight_set = .false.
    allocate (mci_workspace_t :: data)
    select type (data)
    type is (mci_workspace_t)
      data%sampler => sampler
      data%instance => instance
    end select
  select type (rng => mci%rng)
  type is (rng_tao_t)
    if (instance%grids_defined) then

```

```

        call vamp_next_event ( &
            instance%vamp_x, &
            rng%state, &
            instance%grids, &
            vamp_sampling_function, &
            data, &
            phi = phi_trivial)
        instance%vamp_weight = 1
        instance%vamp_weight_set = .true.
    else
        call msg_bug ("vamp: generate event: grids undefined")
    end if
class default
    call msg_fatal ("VAMP event generation: &
        &random-number generator must be TAO")
end select
end select
end subroutine mci_vamp_generate_unweighted_event

```

Rebuild an event, using the `state` input.

```

<MCI vamp: mci vamp: TBP>+≡
    procedure :: rebuild_event => mci_vamp_rebuild_event

<MCI vamp: procedures>+≡
    subroutine mci_vamp_rebuild_event (mci, instance, sampler, state)
        class(mci_vamp_t), intent(inout) :: mci
        class(mci_instance_t), intent(inout) :: instance
        class(mci_sampler_t), intent(inout) :: sampler
        class(mci_state_t), intent(in) :: state
        call msg_bug ("MCI vamp rebuild event not implemented yet")
    !     select type (instance)
    !     type is (mci_vamp_instance_t)
    !         call instance%recall (sampler, state)
    !     end select
    end subroutine mci_vamp_rebuild_event

```

### 13.3.4 Sampler as Workspace

In the full setup, the sampling function requires the process instance object as workspace. We implement this by (i) implementing the process instance as a type extension of the abstract `sampler_t` object used by the MCI implementation and (ii) providing such an object as an extra argument to the sampling function that VAMP can call. To minimize cross-package dependencies, we use an abstract type `vamp_workspace` that VAMP declares and extend this by including a pointer to the `sampler` and `instance` objects. In the body of the sampling function, we dereference this pointer and can then work with the contents.

```

<MCI vamp: types>+≡
    type, extends (vamp_data_t) :: mci_workspace_t
        class(mci_sampler_t), pointer :: sampler => null ()
        class(mci_vamp_instance_t), pointer :: instance => null ()

```

```
end type mci_workspace_t
```

### 13.3.5 Integrator instance

```

<MCI vamp: public>+≡
  public :: mci_vamp_instance_t

<MCI vamp: types>+≡
  type, extends (mci_instance_t) :: mci_vamp_instance_t
    type(mci_vamp_t), pointer :: mci => null ()
    logical :: grids_defined = .false.
    integer :: n_it = 0
    integer :: it = 0
    logical :: pass_complete = .false.
    integer :: n_calls = 0
    integer :: calls = 0
    logical :: it_complete = .false.
    logical :: enable_adapt_grids = .false.
    logical :: enable_adapt_weights = .false.
    logical :: allow_adapt_grids = .false.
    logical :: allow_adapt_weights = .false.
    integer :: n_adapt_grids = 0
    integer :: n_adapt_weights = 0
    logical :: generating_events = .false.
    type(vamp_grids) :: grids
    real(default) :: g = 0
    real(default), dimension(:), allocatable :: gi
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default), dimension(:), allocatable :: vamp_x
    logical :: vamp_weight_set = .false.
    real(default) :: vamp_weight = 0
  contains
    <MCI vamp: mci vamp instance: TBP>
  end type mci_vamp_instance_t

```

Output.

```

<MCI vamp: mci vamp instance: TBP>≡
  procedure :: write => mci_vamp_instance_write

<MCI vamp: procedures>+≡
  subroutine mci_vamp_instance_write (object, unit)
    class(mci_vamp_instance_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(3x,A,ES19.12)") "Integrand = ", object%integrand
    write (u, "(3x,A,ES19.12)") "Weight      = ", object%mci_weight
    if (object%vamp_weight_set) &
      write (u, "(3x,A,ES19.12)") "VAMP wgt  = ", object%vamp_weight
    write (u, "(3x,A,L1)") "adapt grids  = ", object%enable_adapt_grids
    write (u, "(3x,A,L1)") "adapt weights = ", object%enable_adapt_weights
    if (object%grids_defined) then

```

```

        write (u, "(3x,A)") "VAMP grids: defined"
    else
        write (u, "(3x,A)") "VAMP grids: [undefined]"
    end if
    write (u, "(3x,A,I0)") "n_it          = ", object%n_it
    write (u, "(3x,A,I0)") "it           = ", object%it
    write (u, "(3x,A,L1)") "pass complete = ", object%it_complete
    write (u, "(3x,A,I0)") "n_calls       = ", object%n_calls
    write (u, "(3x,A,I0)") "calls        = ", object%calls
    write (u, "(3x,A,L1)") "it complete  = ", object%it_complete
    write (u, "(3x,A,I0)") "n adapt.(g)  = ", object%n_adapt_grids
    write (u, "(3x,A,I0)") "n adapt.(w)  = ", object%n_adapt_weights
    write (u, "(3x,A,L1)") "gen. events  = ", object%generating_events
    write (u, "(3x,A,ES17.10)") "integral = ", object%integral
    write (u, "(3x,A,ES17.10)") "error      = ", object%error
    write (u, "(3x,A)") "weights:"
    do i = 1, size (object%w)
        write (u, "(5x,I0,1x,ES12.5)") i, object%w(i)
    end do
end subroutine mci_vamp_instance_write

```

Write the grids to the specified unit.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: write_grids => mci_vamp_instance_write_grids

<MCI vamp: procedures>+≡
    subroutine mci_vamp_instance_write_grids (object, unit)
        class(mci_vamp_instance_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        if (object%grids_defined) then
            call vamp_write_grids (object%grids, u)
        end if
    end subroutine mci_vamp_instance_write_grids

```

The finalizer is empty.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: final => mci_vamp_instance_final

<MCI vamp: tests>≡
    subroutine mci_vamp_instance_final (object)
        class(mci_vamp_instance_t), intent(inout) :: object
        if (object%grids_defined) then
            call vamp_delete_grids (object%grids)
            object%grids_defined = .false.
        end if
    end subroutine mci_vamp_instance_final

```

Initializer.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: init => mci_vamp_instance_init

```

```

(MCI vamp: procedures)+≡
subroutine mci_vamp_instance_init (mci_instance, mci)
  class(mci_vamp_instance_t), intent(out) :: mci_instance
  class(mci_t), intent(in), target :: mci
  call mci_instance%base_init (mci)
  select type (mci)
  type is (mci_vamp_t)
    mci_instance%mci => mci
    allocate (mci_instance%gi (mci%n_channel))
  end select
end subroutine mci_vamp_instance_init

```

Prepare a new integration pass: write the pass-specific settings to the `instance` object. This should be called initially, together with the `create_grids` procedure, and whenever we start a new integration pass.

Set `reshape` if the number of calls is different than previously (unless it was zero, indicating the first pass).

```

(MCI vamp: mci vamp instance: TBP)+≡
  procedure :: new_pass => mci_vamp_instance_new_pass

(MCI vamp: procedures)+≡
subroutine mci_vamp_instance_new_pass (instance, reshape)
  class(mci_vamp_instance_t), intent(inout) :: instance
  logical, intent(out) :: reshape
  type(pass_t), pointer :: current
  associate (mci => instance%mci)
    current => mci%current_pass
    instance%n_it = current%n_it
    if (instance%n_calls == 0) then
      reshape = .false.
      instance%n_calls = current%n_calls
    else if (instance%n_calls == current%n_calls) then
      reshape = .false.
    else
      reshape = .true.
      instance%n_calls = current%n_calls
    end if
    instance%it = 0
    instance%calls = 0
    instance%enable_adapt_grids = current%adapt_grids
    instance%enable_adapt_weights = current%adapt_weights
    instance%generating_events = .false.
  end associate
end subroutine mci_vamp_instance_new_pass

```

Create a grid set within the `instance` object, using the data of the current integration pass. Also reset counters that track this grid set.

```

(MCI vamp: mci vamp instance: TBP)+≡
  procedure :: create_grids => mci_vamp_instance_create_grids

(MCI vamp: procedures)+≡
subroutine mci_vamp_instance_create_grids (instance)
  class(mci_vamp_instance_t), intent(inout) :: instance

```

```

type (pass_t), pointer :: current
integer, dimension(:), allocatable :: num_div
real(default), dimension(:,:), allocatable :: region
integer :: min_calls
associate (mci => instance%mci)
    current => mci%current_pass
    allocate (num_div (mci%n_dim))
    allocate (region (2, mci%n_dim))
    region(1,:) = 0
    region(2,:) = 1
    num_div = current%n_bins
    instance%n_adapt_grids = 0
    instance%n_adapt_weights = 0
!    call msg_message ("Creating VAMP integration grids:")
!    if (mci%grid_par%use_vamp_equivalences) &
!        call msg_message ("Using phase-space channel equivalences.")
    if (.not. instance%grids_defined) then
        call vamp_create_grids (instance%grids, &
            region, &
            current%n_calls, &
            weights = instance%w, &
            num_div = num_div, &
            stratified = mci%grid_par%stratified)
        instance%grids_defined = .true.
    else
        call msg_bug ("vamp: create grids: grids already defined")
    end if
end associate
end subroutine mci_vamp_instance_create_grids

```

Reset a grid set, so we can start a fresh integration pass. In effect, we delete results of previous integrations, but keep the grid shapes, weights, and variance arrays, so adaptation is still possible. The grids are prepared for a specific number of calls (per iteration) and sampling mode (stratified/importance).

The `vamp_discard_integrals` implementation will reshape the grids only if the argument `num_calls` is present.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: discard_integrals => mci_vamp_instance_discard_integrals
<MCI vamp: procedures>+≡
    subroutine mci_vamp_instance_discard_integrals (instance, reshape)
        class(mci_vamp_instance_t), intent(inout) :: instance
        logical, intent(in) :: reshape
        instance%calls = 0
        instance%integral = 0
        instance%error = 0
        associate (mci => instance%mci)
            if (instance%grids_defined) then
                if (mci%grid_par%use_vamp_equivalences) then
                    if (reshape) then
                        call vamp_discard_integrals (instance%grids, &
                            num_calls = instance%n_calls, &
                            stratified = mci%grid_par%stratified, &
                            eq = mci%equivalences)

```

```

        else
            call vamp_discard_integrals (instance%grids, &
                stratified = mci%grid_par%stratified, &
                eq = mci%equivalences)
        end if
    else
        if (reshape) then
            call vamp_discard_integrals (instance%grids, &
                num_calls = instance%n_calls, &
                stratified = mci%grid_par%stratified)
        else
            call vamp_discard_integrals (instance%grids, &
                stratified = mci%grid_par%stratified)
        end if
    end if
else
    call msg_bug ("vamp: discard integrals: grids undefined")
end if
end associate
end subroutine mci_vamp_instance_discard_integrals

```

After grids are created (with equidistant binning and equal weight), adaptation is redundant. Therefore, we should allow it only after a complete integration step has been performed, calling this.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: allow_adaptation => mci_vamp_instance_allow_adaptation

<MCI vamp: procedures>+≡
    subroutine mci_vamp_instance_allow_adaptation (instance)
        class(mci_vamp_instance_t), intent(inout) :: instance
        instance%allow_adapt_grids = .true.
        instance%allow_adapt_weights = .true.
    end subroutine mci_vamp_instance_allow_adaptation

```

Adapt grids.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: adapt_grids => mci_vamp_instance_adapt_grids

<MCI vamp: procedures>+≡
    subroutine mci_vamp_instance_adapt_grids (instance)
        class(mci_vamp_instance_t), intent(inout) :: instance
        if (instance%enable_adapt_grids .and. instance%allow_adapt_grids) then
            if (instance%grids_defined) then
                call vamp_refine_grids (instance%grids)
                instance%n_adapt_grids = instance%n_adapt_grids + 1
            else
                call msg_bug ("vamp: adapt grids: grids undefined")
            end if
        end if
    end subroutine mci_vamp_instance_adapt_grids

```

Adapt weights. Use the variance array returned by VAMP for recalculating the weight array. The parameter `channel_weights_power` dampens fluctuations.

Note: call thresholds not implemented yet.

```

(MCI vamp: mci vamp instance: TBP)+≡
  procedure :: adapt_weights => mci_vamp_instance_adapt_weights

(MCI vamp: procedures)+≡
  subroutine mci_vamp_instance_adapt_weights (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    real(default) :: w_sum, w_avg_ch
    integer :: n_ch, ch
    logical, dimension(:), allocatable :: mask
    if (instance%enable_adapt_weights .and. instance%allow_adapt_weights) then
      associate (mci => instance%mci)
        if (instance%grids_defined) then
          instance%w = instance%grids%weights &
            * vamp_get_variance (instance%grids%grids) &
            ** mci%grid_par%channel_weights_power
          w_sum = sum (instance%w)
          if (w_sum > 0) then
            instance%w = instance%w / w_sum
            if (mci%n_chain /= 0) then
              allocate (mask (mci%n_channel))
              do ch = 1, mci%n_chain
                mask = mci%chain == ch
                n_ch = count (mask)
                if (n_ch /= 0) then
                  w_avg_ch = sum (instance%w, mask) / n_ch
                  where (mask) instance%w = w_avg_ch
                end if
              end do
            end if
          end if
        end if
      end associate
      instance%n_adapt_weights = instance%n_adapt_weights + 1
    end if
  end subroutine mci_vamp_instance_adapt_weights

```

Integration: sample the VAMP grids. The number of calls etc. are already stored inside the grids. We provide the random-number generator, the sampling function, and a link to the workspace object, which happens to contain a pointer to the sampler object. The sampler object thus becomes the workspace of the sampling function.

Note: in the current implementation, the random-number generator must be the TAO generator. This explicit dependence should be removed from the VAMP implementation.

```

(MCI vamp: mci vamp instance: TBP)+≡
  procedure :: sample_grids => mci_vamp_instance_sample_grids

(MCI vamp: procedures)+≡
  subroutine mci_vamp_instance_sample_grids (instance, rng, sampler, eq)
    class(mci_vamp_instance_t), intent(inout), target :: instance

```



```

class(rng_t), intent(inout) :: rng
class(mci_sampler_t), intent(inout), target :: sampler
type(vamp_equivalences_t), intent(in), optional :: eq
class(vamp_data_t), allocatable :: data
allocate (mci_workspace_t :: data)
select type (data)
type is (mci_workspace_t)
  data%sampler => sampler
  data%instance => instance
end select
select type (rng)
type is (rng_tao_t)
  instance%it = instance%it + 1
  instance%calls = 0
  if (instance%grids_defined) then
    call vamp_sample_grids ( &
      rng%state, &
      instance%grids, &
      vamp_sampling_function, &
      data, &
      1, &
      eq = eq, &
      integral = instance%integral, &
      std_dev = instance%error)
  else
    call msg_bug ("vamp: sample grids: grids undefined")
  end if
class default
  call msg_fatal ("VAMP integration: random-number generator must be TAO")
end select
end subroutine mci_vamp_instance_sample_grids

```

### 13.3.6 Sampling function

The VAMP sampling function has a well-defined interface which we have to implement. The `data` argument allows us to pass pointers to the `sampler` and `instance` objects, so we can access configuration data and fill point-dependent contents within these objects.

The `weights` and `channel` argument must be present in the call.

Note: we would normally declare the `instance` pointer with the concrete type, or just use the `data` component directly. Unfortunately, gfortran 4.6 forgets the inherited base-type methods in that case.

*(MCI vamp: procedures)+≡*

```

function vamp_sampling_function (xi, data, weights, channel, grids) result (f)
  real(default) :: f
  real(default), dimension(:), intent(in) :: xi
  class(vamp_data_t), intent(in) :: data
  real(default), dimension(:), intent(in), optional :: weights
  integer, intent(in), optional :: channel
  type(vamp_grid), dimension(:), intent(in), optional :: grids
  class(mci_instance_t), pointer :: instance
  select type (data)

```

```

type is (mci_workspace_t)
  instance => data%instance
  select type (instance)
    class is (mci_vamp_instance_t)
      instance%calls = instance%calls + 1
    end select
    call instance%evaluate (data%sampler, channel, xi)
    f = instance%get_value ()
  end select
end function vamp_sampling_function

```

This is supposed to be the mapping between integration channels. The VAMP event generating procedures technically require it, but it is meaningless in our setup where all transformations happen inside the sampler object. So, this implementation is trivial:

```

⟨MCI vamp: procedures⟩+=≡
function phi_trivial (xi, channel_dummy) result (x)
  real(default), dimension(:), intent(in) :: xi
  integer, intent(in) :: channel_dummy
  real(default), dimension(size(xi)) :: x
  x = xi
end function phi_trivial

```

### 13.3.7 Integrator instance: evaluation

Here, we compute the multi-channel reweighting factor for the current channel, that accounts for the Jacobians of the transformations from/to all other channels.

The computation of the VAMP probabilities may consume considerable time, therefore we enable parallel evaluation. (Collecting the contributions to `mci%g` is a reduction, which we should also implement via OpenMP.)

Note: inactive channels not implemented yet.

```

⟨MCI vamp: mci vamp instance: TBP⟩+=≡
  procedure :: compute_weight => mci_vamp_instance_compute_weight

⟨MCI vamp: procedures⟩+=≡
subroutine mci_vamp_instance_compute_weight (mci, c)
  class(mci_vamp_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  integer :: i
  mci%selected_channel = c
  !$OMP PARALLEL PRIVATE(i) SHARED(mci)
  !$OMP DO
  do i = 1, mci%mci%n_channel
    mci%gi(i) = vamp_probability (mci%grids%grids(i), mci%x(:,i))
  end do
  !$OMP END DO
  !$OMP END PARALLEL
  mci%g = 0
  do i = 1, mci%mci%n_channel
    mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
  end do

```

```

    if (mci%g /= 0) then
        mci%mci_weight = mci%gi(c) / mci%g
    else
        mci%mci_weight = 0
    end if
end subroutine mci_vamp_instance_compute_weight

```

Record the integrand. Apply the Jacobian weight to get the absolute value. Divide by the channel maximum and by any overall factor to get the value relative to the maximum.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: record_integrand => mci_vamp_instance_record_integrand

<MCI vamp: procedures>+≡
    subroutine mci_vamp_instance_record_integrand (mci, integrand)
        class(mci_vamp_instance_t), intent(inout) :: mci
        real(default), intent(in) :: integrand
        mci%integrand = integrand
    end subroutine mci_vamp_instance_record_integrand

```

Get the event weight. The default routine returns the same value that we would use for integration. This is correct if we select the integration channel according to the channel weight. *vamp\_next\_event* does differently, so we should rather rely on the weight that VAMP returns. This is the value stored in *vamp\_weight*. We override the default TBP accordingly.

```

<MCI vamp: mci vamp instance: TBP>+≡
    procedure :: get_event_weight => mci_vamp_instance_get_event_weight

<MCI vamp: procedures>+≡
    function mci_vamp_instance_get_event_weight (mci) result (value)
        class(mci_vamp_instance_t), intent(in) :: mci
        real(default) :: value
        if (mci%vamp_weight_set) then
            value = mci%vamp_weight
        else
            call msg_bug ("vamp: attempt to read undefined event weight")
        end if
    end function mci_vamp_instance_get_event_weight

```

### 13.3.8 Unit tests

```

<MCI vamp: public>+≡
    public :: mci_vamp_test

<MCI vamp: tests>+≡
    subroutine mci_vamp_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <MCI vamp: execute tests>
    end subroutine mci_vamp_test

```

## Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

In mode 1, the function is  $f(x) = 3x^2$  with integral  $\int_0^1 f(x) dx = 1$  and maximum  $f(1) = 3$ . If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

In mode 2, the function is  $11x^{10}$ , also with integral 1.

```

(MCI vamp: test types)≡
  type, extends (mci_sampler_t) :: test_sampler_1_t
    real(default), dimension(:), allocatable :: x
    real(default) :: val
    integer :: mode = 1
  contains
    (MCI vamp: test sampler 1: TBP)
  end type test_sampler_1_t

```

Output: There is nothing stored inside, so just print an informative line.

```

(MCI vamp: test sampler 1: TBP)≡
  procedure :: write => test_sampler_1_write

(MCI vamp: tests)+≡
  subroutine test_sampler_1_write (object, unit)
    class(test_sampler_1_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    select case (object%mode)
    case (1)
      write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
    case (2)
      write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10"
    case (3)
      write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10 * 2 * cos^2 (2 pi y)"
    end select
  end subroutine test_sampler_1_write

```

Evaluation: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

(MCI vamp: test sampler 1: TBP)+≡
  procedure :: evaluate => test_sampler_1_evaluate

(MCI vamp: tests)+≡
  subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_1_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (allocated (sampler%x)) deallocate (sampler%x)
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in

```

```

select case (sampler%mode)
case (1)
  sampler%val = 3 * x_in(1) ** 2
case (2)
  sampler%val = 11 * x_in(1) ** 10
case (3)
  sampler%val = 11 * x_in(1) ** 10 * 2 * cos (twopi * x_in(2)) ** 2
end select
call sampler%fetch (val, x, f)
end subroutine test_sampler_1_evaluate

```

Rebuild: compute all but the function value.

```

⟨MCI vamp: test sampler 1: TBP⟩+≡
  procedure :: rebuild => test_sampler_1_rebuild

⟨MCI vamp: tests⟩+≡
  subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_1_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (allocated (sampler%x)) deallocate (sampler%x)
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    sampler%val = val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_rebuild

```

Extract the results.

```

⟨MCI vamp: test sampler 1: TBP⟩+≡
  procedure :: fetch => test_sampler_1_fetch

⟨MCI vamp: tests⟩+≡
  subroutine test_sampler_1_fetch (sampler, val, x, f)
    class(test_sampler_1_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_fetch

```

## Two-channel, two dimension test sampler

This sampler implements the function

$$f(x, y) = 4 \sin^2(\pi x) \sin^2(\pi y) + 2 \sin^2(\pi v) \quad (13.11)$$

where

$$x = u^v \qquad u = xy \qquad (13.12)$$

$$y = u^{(1-v)} \qquad v = \frac{1}{2} \left( 1 + \frac{\log(x/y)}{\log xy} \right) \qquad (13.13)$$

Each term contributes 1 to the integral. The first term in the function is peaked along a cross aligned to the coordinates  $x$  and  $y$ , while the second term is peaked along the diagonal  $x = y$ .

The Jacobian is

$$\frac{\partial(x, y)}{\partial(u, v)} = |\log u| \qquad (13.14)$$

```

<MCI vamp: test types>+≡
  type, extends (mci_sampler_t) :: test_sampler_2_t
    real(default), dimension(:,:), allocatable :: x
    real(default), dimension(:), allocatable :: f
    real(default) :: val
  contains
    <MCI vamp: test sampler 2: TBP>
  end type test_sampler_2_t

```

Output: There is nothing stored inside, so just print an informative line.

```

<MCI vamp: test sampler 2: TBP>≡
  procedure :: write => test_sampler_2_write

<MCI vamp: tests>+≡
  subroutine test_sampler_2_write (object, unit)
    class(test_sampler_2_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Two-channel test sampler 2"
  end subroutine test_sampler_2_write

```

Kinematics: compute  $x$  and Jacobians, given the input parameter array.

```

<MCI vamp: test sampler 2: TBP>+≡
  procedure :: compute => test_sampler_2_compute

<MCI vamp: tests>+≡
  subroutine test_sampler_2_compute (sampler, c, x_in)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default) :: xx, yy, uu, vv
    if (.not. allocated (sampler%x)) &
      allocate (sampler%x (size (x_in), 2))
    if (.not. allocated (sampler%f)) &
      allocate (sampler%f (2))
    select case (c)
    case (1)
      xx = x_in(1)
      yy = x_in(2)
      uu = xx * yy

```

```

        vv = (1 + log (xx/yy) / log (xx*yy)) / 2
    case (2)
        uu = x_in(1)
        vv = x_in(2)
        xx = uu ** vv
        yy = uu ** (1 - vv)
    end select
    sampler%val = (2 * sin (pi * xx) * sin (pi * yy)) ** 2 &
        + 2 * sin (pi * vv) ** 2
    sampler%f(1) = 1
    sampler%f(2) = abs (log (uu))
    sampler%x(:,1) = [xx, yy]
    sampler%x(:,2) = [uu, vv]
end subroutine test_sampler_2_compute

```

Evaluation: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

⟨MCI vamp: test sampler 2: TBP⟩+≡
    procedure :: evaluate => test_sampler_2_evaluate

⟨MCI vamp: tests⟩+≡
    subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
        class(test_sampler_2_t), intent(inout) :: sampler
        integer, intent(in) :: c
        real(default), dimension(:), intent(in) :: x_in
        real(default), intent(out) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        call sampler%compute (c, x_in)
        call sampler%fetch (val, x, f)
    end subroutine test_sampler_2_evaluate

```

Rebuild: compute all but the function value.

```

⟨MCI vamp: test sampler 2: TBP⟩+≡
    procedure :: rebuild => test_sampler_2_rebuild

⟨MCI vamp: tests⟩+≡
    subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
        class(test_sampler_2_t), intent(inout) :: sampler
        integer, intent(in) :: c
        real(default), dimension(:), intent(in) :: x_in
        real(default), intent(in) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        call sampler%compute (c, x_in)
        x = sampler%x
        f = sampler%f
    end subroutine test_sampler_2_rebuild

```

Extract the results.

```

⟨MCI vamp: test sampler 2: TBP⟩+≡
    procedure :: fetch => test_sampler_2_fetch

```

```

<MCI vamp: tests>+=
  subroutine test_sampler_2_fetch (sampler, val, x, f)
    class(test_sampler_2_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_2_fetch

```

### Two-channel, one dimension test sampler

This sampler implements the function

$$f(x, y) = a * 5x^4 + b * 5(1 - x)^4 \quad (13.15)$$

Each term contributes 1 to the integral, multiplied by  $a$  or  $b$ , respectively. The first term is peaked at  $x = 1$ , the second one at  $x = 0$ .

We implement the two mappings

$$x = u^{1/5} \quad \text{and} \quad x = 1 - v^{1/5}, \quad (13.16)$$

with Jacobians

$$\frac{\partial(x)}{\partial(u)} = u^{-4/5}/5 \quad \text{and} \quad v^{-4/5}/5, \quad (13.17)$$

respectively. The first mapping concentrates points near  $x = 1$ , the second one near  $x = 0$ .

```

<MCI vamp: test types>+=
  type, extends (mci_sampler_t) :: test_sampler_3_t
    real(default), dimension(:,:), allocatable :: x
    real(default), dimension(:), allocatable :: f
    real(default) :: val
    real(default) :: a = 1
    real(default) :: b = 1
  contains
    <MCI vamp: test sampler 3: TBP>
  end type test_sampler_3_t

```

Output: display  $a$  and  $b$

```

<MCI vamp: test sampler 3: TBP>=
  procedure :: write => test_sampler_3_write
<MCI vamp: tests>+=
  subroutine test_sampler_3_write (object, unit)
    class(test_sampler_3_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Two-channel test sampler 3"
    write (u, "(3x,A,F5.2)") "a = ", object%a
    write (u, "(3x,A,F5.2)") "b = ", object%b
  end subroutine test_sampler_3_write

```



Kinematics: compute  $x$  and Jacobians, given the input parameter array.

```

(MCI vamp: test sampler 3: TBP)+≡
  procedure :: compute => test_sampler_3_compute

(MCI vamp: tests)+≡
  subroutine test_sampler_3_compute (sampler, c, x_in)
    class(test_sampler_3_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default) :: u, v, xx
    if (.not. allocated (sampler%x)) &
      allocate (sampler%x (size (x_in), 2))
    if (.not. allocated (sampler%f)) &
      allocate (sampler%f (2))
    select case (c)
    case (1)
      u = x_in(1)
      xx = u ** 0.2_default
      v = (1 - xx) ** 5._default
    case (2)
      v = x_in(1)
      xx = 1 - v ** 0.2_default
      u = xx ** 5._default
    end select
    sampler%val = sampler%a * 5 * xx ** 4 + sampler%b * 5 * (1 - xx) ** 4
    sampler%f(1) = 0.2_default * u ** (-0.8_default)
    sampler%f(2) = 0.2_default * v ** (-0.8_default)
    sampler%x(:,1) = [u]
    sampler%x(:,2) = [v]
  end subroutine test_sampler_3_compute

```

Evaluation: compute the function value. The output  $x$  parameter (only one channel) is identical to the input  $x$ , and the Jacobian is 1.

```

(MCI vamp: test sampler 3: TBP)+≡
  procedure :: evaluate => test_sampler_3_evaluate

(MCI vamp: tests)+≡
  subroutine test_sampler_3_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_3_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_3_evaluate

```

Rebuild: compute all but the function value.

```

(MCI vamp: test sampler 3: TBP)+≡
  procedure :: rebuild => test_sampler_3_rebuild

```

```

<MCI vamp: tests>+=
subroutine test_sampler_3_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_3_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call sampler%compute (c, x_in)
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_3_rebuild

```

Extract the results.

```

<MCI vamp: test sampler 3: TBP>+=
  procedure :: fetch => test_sampler_3_fetch

<MCI vamp: tests>+=
subroutine test_sampler_3_fetch (sampler, val, x, f)
  class(test_sampler_3_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_3_fetch

```

## One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

Note: We would like to check the precise contents of the grid allocated during integration, but the output format for reals is very long (for good reasons), so the last digits in the grid content display are numerical noise. So, we just check the integration results.

```

<MCI vamp: execute tests>=
  call test (mci_vamp_1, "mci_vamp_1", &
    "one-dimensional integral", &
    u, results)

<MCI vamp: tests>+=
subroutine mci_vamp_1 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  real(default) :: integrand

  write (u, "(A)")  "* Test output: mci_vamp_1"
  write (u, "(A)")  "* Purpose: integrate function in one dimension &

```

```

        &(single channel)"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 1)
select type (mci)
type is (mci_vamp_t)
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_1_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass ()
end select
call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

!     write (u, "(A)")
!     write (u, "(A)")  "* Grids"
!     write (u, "(A)")
!

```

```

!      select type (mci_instance)
!      type is (mci_vamp_instance_t)
!      call mci_instance%write_grids (u)
!      end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_1"

end subroutine mci_vamp_1

```

## Multiple iterations

Construct an integrator and use it for a one-dimensional sampler. Integrate with five iterations without grid adaptation.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_2, "mci_vamp_2", &
    "multiple iterations", &
    u, results)

<MCI vamp: tests>+≡
  subroutine mci_vamp_2 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    type(vamp_equivalences_t) :: eq
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_vamp_2"
    write (u, "(A)")  "* Purpose: integrate function in one dimension &
      &(single channel)"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 1)
    select type (mci)
    type is (mci_vamp_t)
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)

```

```

call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_1_t :: sampler)
select type (sampler)
type is (test_sampler_1_t)
    sampler%mode = 2
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .false.)
end select
call mci%integrate (mci_instance, sampler, 3, 100)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_2"

end subroutine mci_vamp_2

```

## Grid adaptation

Construct an integrator and use it for a one-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

```

<MCI vamp: execute tests>+≡
    call test (mci_vamp_3, "mci_vamp_3", &
        "grid adaptation", &
        u, results)

<MCI vamp: tests>+≡
    subroutine mci_vamp_3 (u)
        integer, intent(in) :: u

```

```

type(grid_parameters_t) :: grid_par
type(vamp_equivalences_t) :: eq
class(mci_t), allocatable, target :: mci
class(mci_instance_t), pointer :: mci_instance => null ()
class(mci_sampler_t), allocatable :: sampler
class(rng_t), allocatable :: rng
real(default) :: integrand

write (u, "(A)")  "* Test output: mci_vamp_3"
write (u, "(A)")  "* Purpose: integrate function in one dimension &
    &(single channel)"
write (u, "(A)")  "*          and adapt grid"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 1)
select type (mci)
type is (mci_vamp_t)
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_1_t :: sampler)
select type (sampler)
type is (test_sampler_1_t)
    sampler%mode = 2
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 100)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

```

```

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_3"

end subroutine mci_vamp_3

```

## Two-dimensional integral

Construct an integrator and use it for a two-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

```

<MCI vamp: execute tests>+≡
call test (mci_vamp_4, "mci_vamp_4", &
  "two-dimensional integration", &
  u, results)

<MCI vamp: tests>+≡
subroutine mci_vamp_4 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  type(vamp_equivalences_t) :: eq
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  real(default) :: integrand

  write (u, "(A)")  "* Test output: mci_vamp_4"
  write (u, "(A)")  "* Purpose: integrate function in two dimensions &
    &(single channel)"
  write (u, "(A)")  "*           and adapt grid"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (2, 1)
  select type (mci)
  type is (mci_vamp_t)
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

```

```

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_1_t :: sampler)
select type (sampler)
type is (test_sampler_1_t)
  sampler%mode = 3
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_4"

end subroutine mci_vamp_4

```

## Two-channel integral

Construct an integrator and use it for a two-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_5, "mci_vamp_5", &
    "two-dimensional integration", &
    u, results)

<MCI vamp: tests>+≡
  subroutine mci_vamp_5 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par

```



```

type(vamp_equivalences_t) :: eq
class(mci_t), allocatable, target :: mci
class(mci_instance_t), pointer :: mci_instance => null ()
class(mci_sampler_t), allocatable :: sampler
class(rng_t), allocatable :: rng
real(default) :: integrand

write (u, "(A)")  "* Test output: mci_vamp_5"
write (u, "(A)")  "* Purpose: integrate function in two dimensions &
                    &(two channels)"
write (u, "(A)")  "*               and adapt grid"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_2_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

```

```

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_5"

end subroutine mci_vamp_5

```

## Weight adaptation

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between weight adaptations.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_6, "mci_vamp_6", &
    "weight adaptation", &
    u, results)

<MCI vamp: tests>+≡
  subroutine mci_vamp_6 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    type(vamp_equivalences_t) :: eq
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_vamp_6"
    write (u, "(A)")  "* Purpose: integrate function in one dimension &
      &(two channels)"
    write (u, "(A)")  "*           and adapt weights"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 2)
    select type (mci)
    type is (mci_vamp_t)
      grid_par%stratified = .false.
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)

```

```

call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
    sampler%a = 0.9_default
    sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Re-initialize with chained channels"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
call mci%declare_chains ([1,1])
select type (mci)
type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)

```

```

call mci_instance%init (mci)

write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%add_pass (adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_6"

end subroutine mci_vamp_6

```

## Equivalences

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations. Apply an equivalence between the two channels, so the binning of the two channels is forced to coincide. Compare this with the behavior without equivalences.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_7, "mci_vamp_7", &
    "use channel equivalences", &
    u, results)

<MCI vamp: tests>+≡
  subroutine mci_vamp_7 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    type(phs_channel_t), dimension(:), allocatable :: channel
    class(rng_t), allocatable :: rng
    real(default), dimension(:,:), allocatable :: x
    integer :: u_grid, iostat, i, div, ch
    character(16) :: buffer

```

```

write (u, "(A)")  "* Test output: mci_vamp_7"
write (u, "(A)")  "* Purpose: check effect of channel equivalences"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
    sampler%a = 0.7_default
    sampler%b = 0.3_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
    &adapt grids"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 2, 1000)

call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write grids and extract binning"
write (u, "(A)")

u_grid = free_unit ()
open (u_grid, status = "scratch", action = "readwrite")
select type (mci_instance)
type is (mci_vamp_instance_t)
    call vamp_write_grids (mci_instance%grids, u_grid)

```

```

end select
rewind (u_grid)
allocate (x (0:20, 2))
do div = 1, 2
  FIND_BINS1: do
    read (u_grid, "(A)") buffer
    if (trim (adjustl (buffer)) == "begin d%x") then
      do
        read (u_grid, *, iostat = iostat) i, x(i,div)
        if (iostat /= 0) exit FIND_BINS1
      end do
    end if
  end do FIND_BINS1
end do
close (u_grid)

write (u, "(1x,A,L1)") "Equal binning in both channels = ", &
  all (x(:,1) == x(:,2))
deallocate (x)

write (u, "(A)")
write (u, "(A)") "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)") "* Re-initialize integrator, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
  grid_par%stratified = .false.
  grid_par%use_vamp_equivalences = .true.
  call mci%set_grid_parameters (grid_par)
end select

write (u, "(A)") "* Define equivalences"
write (u, "(A)")

allocate (channel (2))
do ch = 1, 2
  allocate (channel(ch)%eq (2))
  do i = 1, 2
    associate (eq => channel(ch)%eq(i))
      call eq%init (1)
      eq%c = i
      eq%perm = [1]
      eq%mode = [0]
    end associate
  end do
end do

```

```

        end do
        write (u, "(1x,I0,':')", advance = "no")  ch
        call channel(ch)%write (u)
    end do
    call mci%declare_equivalences (channel, dim_offset = 0)

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
        &adapt grids"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp_t)
        call mci%add_pass (adapt_grids = .true.)
    end select
    call mci%integrate (mci_instance, sampler, 2, 1000)

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write grids and extract binning"
    write (u, "(A)")

    u_grid = free_unit ()
    open (u_grid, status = "scratch", action = "readwrite")
    select type (mci_instance)
    type is (mci_vamp_instance_t)
        call vamp_write_grids (mci_instance%grids, u_grid)
    end select
    rewind (u_grid)
    allocate (x (0:20, 2))
    do div = 1, 2
        FIND_BINS2: do
            read (u_grid, "(A)")  buffer
            if (trim (adjustl (buffer)) == "begin d%x") then
                do
                    read (u_grid, *, iostat = iostat)  i, x(i,div)
                    if (iostat /= 0)  exit FIND_BINS2
                end do
            end if
        end do FIND_BINS2
    end do
    close (u_grid)

    write (u, "(1x,A,L1)")  "Equal binning in both channels = ", &
        all (x(:,1) == x(:,2))
    deallocate (x)

```

```

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_7"

end subroutine mci_vamp_7

```

## Multiple passes

Integrate with three passes and different settings for weight and grid adaptation.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_8, "mci_vamp_8", &
    "integration passes", &
    u, results)

<MCI vamp: tests>+≡
subroutine mci_vamp_8 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  type(vamp_equivalences_t) :: eq
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  real(default) :: integrand

  write (u, "(A)")  "* Test output: mci_vamp_8"
  write (u, "(A)")  "* Purpose: integrate function in one dimension &
    &(two channels)"
  write (u, "(A)")  "*           in three passes"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 2)
  select type (mci)
  type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

```



```

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
    sampler%a = 0.9_default
    sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid and weight adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .true., adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate without adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%add_pass ()
end select

```

```

call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_8"

end subroutine mci_vamp_8

```

## Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate and generate a weighted event.

```

<MCI vamp: execute tests>+≡
  call test (mci_vamp_9, "mci_vamp_9", &
    "weighted event", &
    u, results)

<MCI vamp: tests>+≡
  subroutine mci_vamp_9 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    type(vamp_equivalences_t) :: eq
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_vamp_9"
    write (u, "(A)")  "* Purpose: integrate function in two dimensions &
      &(two channels)"
    write (u, "(A)")  "* and generate a weighted event"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_vamp_t)

```

```

        grid_par%stratified = .false.
        grid_par%use_vamp_equivalences = .false.
        call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_2_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
    write (u, "(A)")

    call mci%add_pass ()
    call mci%integrate (mci_instance, sampler, 1, 1000)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate a weighted event"
    write (u, "(A)")

    call mci%init_simulation (mci_instance)
    call mci%generate_weighted_event (mci_instance, sampler)

    write (u, "(1x,A)")  "MCI instance:"
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci%final_simulation (mci_instance)
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_9"

end subroutine mci_vamp_9

```

## Grids I/O

Construct an integrator and allocate grids. Write grids to file, read them in again and compare.

```

<MCI vamp: execute tests>+≡
    call test (mci_vamp_10, "mci_vamp_10", &
        "grids I/O", &

```

```

        u, results)
<MCI vamp: tests>+=
subroutine mci_vamp_10 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    type(string_t) :: file1, file2
    character(80) :: buffer1, buffer2
    integer :: u1, u2, iostat1, iostat2
    logical :: equal, success

    write (u, "(A)")  "* Test output: mci_vamp_10"
    write (u, "(A)")  "* Purpose: write and read VAMP grids"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_vamp_t)
        grid_par%stratified = .false.
        grid_par%use_vamp_equivalences = .false.
        call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    mci%md5sum = "1234567890abcdef1234567890abcdef"

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_2_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
    write (u, "(A)")

    call mci%add_pass ()
    call mci%integrate (mci_instance, sampler, 1, 1000)

    write (u, "(A)")  "* Write grids to file"
    write (u, "(A)")

    file1 = "mci_vamp_10.1"
    select type (mci)

```

```

type is (mci_vamp_t)
  call mci%set_grid_filename (file1)
  call mci%write_grids (mci_instance)
end select

call mci_instance%final ()
call mci%final ()
deallocate (mci)

write (u, "(A)")  "* Read grids from file"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
  call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

mci%md5sum = "1234567890abcdef1234567890abcdef"

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

select type (mci)
type is (mci_vamp_t)
  call mci%set_grid_filename (file1)
  call mci%add_pass ()
  call mci%current_pass%configure (1, 1000, &
    mci%min_calls, &
    mci%grid_par%min_bins, mci%grid_par%max_bins)
  call mci%read_grids (mci_instance, success)
  call mci%compute_md5sum ()
end select
write (u, "(1x,A,L1)")  "success = ", success

write (u, "(A)")
write (u, "(A)")  "* Write grids again"
write (u, "(A)")

file2 = "mci_vamp_10.2"
select type (mci)
type is (mci_vamp_t)
  call mci%set_grid_filename (file2)
  call mci%write_grids (mci_instance)
end select

u1 = free_unit ()
open (u1, file = char (file1) // ".vg", action = "read", status = "old")
u2 = free_unit ()

```

```

open (u2, file = char (file2) // ".vg", action = "read", status = "old")

equal = .true.
iostat1 = 0
iostat2 = 0
do while (equal .and. iostat1 == 0 .and. iostat2 == 0)
    read (u1, "(A)", iostat = iostat1) buffer1
    read (u2, "(A)", iostat = iostat2) buffer2
    equal = buffer1 == buffer2 .and. iostat1 == iostat2
end do
close (u1)
close (u2)

if (equal) then
    write (u, "(1x,A)") "Success: grid files are identical"
else
    write (u, "(1x,A)") "Failure: grid files differ"
end if

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_10"

end subroutine mci_vamp_10

```

## Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate, write grids, and generate a weighted event using the grids from file.

```

<MCI vamp: execute tests>+≡
    call test (mci_vamp_11, "mci_vamp_11", &
        "weighted events with grid I/0", &
        u, results)

<MCI vamp: tests>+≡
    subroutine mci_vamp_11 (u)
        integer, intent(in) :: u
        type(grid_parameters_t) :: grid_par
        type(vamp_equivalences_t) :: eq
        class(mci_t), allocatable, target :: mci
        class(mci_instance_t), pointer :: mci_instance => null ()
        class(mci_sampler_t), allocatable :: sampler
        class(rng_t), allocatable :: rng
        real(default) :: integrand

        write (u, "(A)")  "* Test output: mci_vamp_11"
        write (u, "(A)")  "* Purpose: integrate function in two dimensions &

```

```

        &(two channels)"
write (u, "(A)")  "*"                and generate a weighted event"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
    call mci%set_grid_filename (var_str ("mci_vamp_11"))
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_2_t :: sampler)

write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

call mci%add_pass ()
call mci%integrate (mci_instance, sampler, 1, 1000)

write (u, "(A)")  "* Reset instance"
write (u, "(A)")

call mci_instance%final ()
call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Generate a weighted event"
write (u, "(A)")

call mci%init_simulation (mci_instance)
call mci%generate_weighted_event (mci_instance, sampler)

write (u, "(A)")  "* Cleanup"

call mci%final_simulation (mci_instance)
call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_11"

```

```
end subroutine mci_vamp_11
```

## Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate, write grids, and generate a weighted event using the grids from file.

```
(MCI vamp: execute tests)+≡
  call test (mci_vamp_12, "mci_vamp_12", &
    "unweighted events with grid I/O", &
    u, results)

(MCI vamp: tests)+≡
  subroutine mci_vamp_12 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    type(vamp_equivalences_t) :: eq
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    real(default) :: integrand

    write (u, "(A)")  "* Test output: mci_vamp_12"
    write (u, "(A)")  "* Purpose: integrate function in two dimensions &
      &(two channels)"
    write (u, "(A)")  "*           and generate an unweighted event"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_vamp_t)
      grid_par%stratified = .false.
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
      call mci%set_grid_filename (var_str ("mci_vamp_12"))
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_2_t :: sampler)

    write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
    write (u, "(A)")
```



```

call mci%add_pass ()
call mci%integrate (mci_instance, sampler, 1, 1000)

write (u, "(A)")  "* Reset instance"
write (u, "(A)")

call mci_instance%final ()
call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Generate an unweighted event"
write (u, "(A)")

call mci%init_simulation (mci_instance)
call mci%generate_unweighted_event (mci_instance, sampler)

write (u, "(1x,A)")  "MCI instance:"
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci%final_simulation (mci_instance)
call mci_instance%final ()
call mci%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_12"

end subroutine mci_vamp_12

```

## Update integration results

Compare two mci objects; match the two and update the first if successful.

```

<MCI vamp: execute tests>+≡
call test (mci_vamp_13, "mci_vamp_13", &
  "updating integration results", &
  u, results)

<MCI vamp: tests>+≡
subroutine mci_vamp_13 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  type(vamp_equivalences_t) :: eq
  class(mci_t), allocatable, target :: mci, mci_ref
  logical :: success

  write (u, "(A)")  "* Test output: mci_vamp_13"
  write (u, "(A)")  "* Purpose: match and update integrators"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator with no passes"

```

```

write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
    grid_par%stratified = .false.
    grid_par%use_vamp_equivalences = .false.
    call mci%set_grid_parameters (grid_par)
end select
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize reference"
write (u, "(A)")

allocate (mci_vamp_t :: mci_ref)
call mci_ref%set_dimensions (2, 2)
select type (mci_ref)
type is (mci_vamp_t)
    call mci_ref%set_grid_parameters (grid_par)
end select

select type (mci_ref)
type is (mci_vamp_t)
    call mci_ref%add_pass (adapt_grids = .true.)
    call mci_ref%current_pass%configure (2, 1000, 0, 1, 5)
    mci_ref%current_pass%calls = [77, 77]
    mci_ref%current_pass%integral = [1.23_default, 3.45_default]
    mci_ref%current_pass%error = [0.23_default, 0.45_default]
    mci_ref%current_pass%integral_defined = .true.

    call mci_ref%add_pass ()
    call mci_ref%current_pass%configure (2, 2000, 0, 1, 7)
    mci_ref%current_pass%calls = [99, 0]
    mci_ref%current_pass%integral = [7.89_default, 0._default]
    mci_ref%current_pass%error = [0.89_default, 0._default]
    mci_ref%current_pass%integral_defined = .true.
end select

call mci_ref%write(u)

write (u, "(A)")
write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
    call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

```

```

write (u, "(A)")
write (u, "(A)")  "* Add pass to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%add_pass (adapt_grids = .true.)
  call mci%current_pass%configure (2, 1000, 0, 1, 5)
  mci%current_pass%calls = [77, 77]
  mci%current_pass%integral = [1.23_default, 3.45_default]
  mci%current_pass%error = [0.23_default, 0.45_default]
  mci%current_pass%integral_defined = .true.
end select

write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add pass to integrator, wrong parameters"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%add_pass ()
  call mci%current_pass%configure (2, 1000, 0, 1, 7)
end select

write (u, "(A)")  "* Update integrator (should fail)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset and add passes to integrator"
write (u, "(A)")

select type (mci)

```

```

type is (mci_vamp_t)
  call mci%reset ()
  call mci%add_pass (adapt_grids = .true.)
  call mci%current_pass%configure (2, 1000, 0, 1, 5)
  mci%current_pass%calls = [77, 77]
  mci%current_pass%integral = [1.23_default, 3.45_default]
  mci%current_pass%error = [0.23_default, 0.45_default]
  mci%current_pass%integral_defined = .true.

  call mci%add_pass ()
  call mci%current_pass%configure (2, 2000, 0, 1, 7)
end select

write (u, "(A)")  "* Update integrator (should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Update again (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add extra result to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  mci%current_pass%calls(2) = 1234
end select

write (u, "(A)")  "* Update integrator (should fail)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
  call mci%update_from_ref (mci_ref, success)
end select

```

```

write (u, "(1x,A,L1)") "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci%final ()
call mci_ref%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: mci_vamp_13"

end subroutine mci_vamp_13

```

## Chapter 14

# Process Libraries

This chapter consists of two modules which organize and interface the interaction matrix elements for a variety of elementary processes.

**prclib\_interfaces** This module deals with matrix-element code which is accessible via external libraries (Fortran libraries or generic C-compatible libraries) and must either be generated by the program or provided by the user explicitly.

The module defines and uses an abstract type `prc_writer_t` and two abstract extensions, one for a Fortran module and one for a C-compatible library. The implementation provides the specific methods for writing the appropriate parts in external matrix element code.

**prc\_core\_def** This module defines the abstract types `prc_core_def_t` and `prc_driver_t`. The implementation of the former provides the configuration for processes of a certain class, while the latter accesses the corresponding matrix element, in particular those generated by the appropriate `prc_writer_t` object.

**process\_libraries** This module combines the functionality of the previous module with the means for holding processes definitions (the internal counterpart of appropriate declarations in the user interface), for handling matrix elements which do not need external code, and for accessing the matrix elements by the procedures for matrix-element evaluation, integration and event generation.

**test\_me** This module provides a test implementation for the abstract types in the `prc_core_def` module. The implementation is intended for self-tests of several later modules. The implementation is internal, i.e., no external code has is generated.

All data structures which are specific for a particular way of generating code or evaluating matrix element are kept abstract and thus generic. Later modules such as `prc_omega` provide implementations, in the form of type extensions for the various abstract types.

## 14.1 Process library interface

The module `prclib_interfaces` handles external matrix-element code.

### 14.1.1 Overview

The top-level data structure is the `prclib_driver_t` data type. The associated type-bound procedures deal with the generation of external code, compilation and linking, and accessing the active external library.

An object of type `prclib_driver_t` consists of the following parts:

1. Metadata that identify name and status of the library driver, etc.
2. An array of process records (`prclib_driver_record_t`), one for each external matrix element.
3. A record of type `dlaccess_t` which handles the operating-system part of linking a dynamically loadable library.
4. A collection of procedure pointers which have a counterpart in the external library interface. Given the unique identifier of a matrix element, the procedures retrieve generic matrix-element information such as the particle content and helicity combination tables. There is also a procedure which returns pointers to the more specific procedures that a matrix element provides, called *features*.

The process records of type `prclib_driver_record_t` handle the individual matrix elements. Each record identifies a process by name (`id`), names the physics model to be loaded for this process, lists the features that the associated matrix-element code provides, and holds a `writer` object which handles all operations that depend on the process type. The numbering of process records is identical to the numbering of matrix-element codes in the external library.

The writer object is of abstract type `prc_writer_t`. The module defines two basic, also abstract, extensions: `prc_writer_f_module_t` and `prc_writer_c_lib_t`. The first version is for matrix-element code that is available in form of Fortran modules. The writer contains type-bound procedures which create appropriate `use` directives and C-compatible wrapper functions for the given set of Fortran modules and their features. The second version is for matrix-element code that is available in form of a C-compatible library (this includes Fortran libraries with proper C bindings). The writer needs not write wrapper function, but explicit interface blocks for the matrix-element features.

Each matrix-element variant is encoded in an appropriate extension of `prc_writer_t`. For instance, `O'MEGA` matrix elements provide an implementation `omega_writer_t` which extends `prc_writer_f_module_t`.

### 14.1.2 Workflow

We expect that the functionality provided by this module is called in the following order:

1. The caller initializes the `prclib_driver_t` object and fills the array of `prclib_record_t` entries with the appropriate process data and process-specific writer objects.

2. It calls the `generate_makefile` method to set up an appropriate makefile in the current directory. The makefile will handle source generation, compilation and linking both for the individual matrix elements (unless this has to be done manually) and for the common external driver code which interfaces those matrix element.
3. The `generate_driver_code` writes the common driver as source code to file.
4. The methods `make_source`, `make_compile`, and `make_link` individually perform the corresponding steps in building the library. Wherever possible, they simply use the generated makefile. By calling `make`, we make sure that we can avoid unnecessary recompilation. For the compilation and linking steps, the makefile will employ `libtool`.
5. The `load` method loads the library procedures into the corresponding procedure pointers, using the `dlopen` mechanism via the `dlaccess` subobject.

### 14.1.3 The module

```

<prclib_interfaces.f90>≡
  <File header>

  module prclib_interfaces

    use iso_c_binding !NODEP!
    use kinds !NODEP!
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use limits, only: TAB !NODEP!
    use unit_tests
    use os_interface

    <Standard module head>

    <Prclib interfaces: public>

    <Prclib interfaces: types>

    <Prclib interfaces: interfaces>

    <Prclib interfaces: test types>

    contains

    <Prclib interfaces: procedures>

    <Prclib interfaces: tests>

  end module prclib_interfaces

```



### 14.1.4 Writers

External matrix element code provides externally visible procedures, which we denote as *features*. The features consist of informational subroutines and functions which are mandatory (universal features) and matrix-element specific subroutines and functions (specific features). The driver interfaces the generic features directly, while it returns the specific features in form of `bind(C)` procedure pointers to the caller. For instance, function `n_in` is generic, while the matrix matrix-element value itself is specific.

To implement these tasks, the driver needs `use` directives for Fortran module procedures, interface blocks for other external stuff, wrapper code, and Makefile snippets.

#### Generic writer

In the `prc_writer_t` data type, we collect the procedures which implement the writing tasks. The type is abstract. The concrete implementations are defined by an extension which is specific for the process type.

The MD5 sum stored here should be the MD5 checksum of the current process component, which can be calculated once the process is configured completely. It can be used by implementations which work with external files, such as O'MEGA.

```
<Prclib interfaces: public>≡
  public :: prc_writer_t

<Prclib interfaces: types>≡
  type, abstract :: prc_writer_t
    character(32) :: md5sum = ""
    contains
    <Prclib interfaces: prc writer: TBP>
  end type prc_writer_t
```

In any case, it is useful to have a string representation of the writer type. This must be implemented by all extensions.

```
<Prclib interfaces: prc writer: TBP>≡
  procedure(get_const_string), nopass, deferred :: type_name

<Prclib interfaces: interfaces>≡
  abstract interface
    function get_const_string () result (string)
      import
      type(string_t) :: string
    end function get_const_string
  end interface
```

Return the name of a procedure that implements a given feature, as it is provided by the external matrix-element code. For a reasonable default, we take the feature name unchanged.

```
<Prclib interfaces: prc writer: TBP>+≡
  procedure, nopass :: get_procname => prc_writer_get_procname
```

```

<Prclib interfaces: procedures>≡
function prc_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  name = feature
end function prc_writer_get_procname

```

Return the name of a procedure that implements a given feature with the bind(C) property, so it can be accessed via a C procedure pointer and handled by dlopen. We need this for all special features of a matrix element, since the interface has to return a C function pointer for it. For a default implementation, we prefix the external procedure name by the process ID.

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure :: get_c_procname => prc_writer_get_c_procname

<Prclib interfaces: procedures>+≡
function prc_writer_get_c_procname (writer, id, feature) result (name)
  class(prc_writer_t), intent(in) :: writer
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = id // "_" // feature
end function prc_writer_get_c_procname

```

Common signature of code-writing procedures. The procedure may use the process ID, and the feature name. (Not necessarily all of them.)

```

<Prclib interfaces: interfaces>+≡
  abstract interface
    subroutine write_code_file (writer, id)
      import
      class(prc_writer_t), intent(in) :: writer
      type(string_t), intent(in) :: id
    end subroutine write_code_file
  end interface

  abstract interface
    subroutine write_code (writer, unit, id)
      import
      class(prc_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
    end subroutine write_code
  end interface

  abstract interface
    subroutine write_code_os (writer, unit, id, os_data)
      import
      class(prc_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(os_data_t), intent(in) :: os_data
    end subroutine write_code_os
  end interface

```

```

abstract interface
  subroutine write_feature_code (writer, unit, id, feature)
    import
    class(prc_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
  end subroutine write_feature_code
end interface

```

There must be a procedure which writes an interface block for a given feature. If the external matrix element is implemented as a Fortran module, this is required only for the specific features which are returned as procedure pointers.

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_feature_code), deferred :: write_interface

```

There must also be a procedure which writes Makefile code which is specific for the current process, but not the feature.

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_code_os), deferred :: write_makefile_code

```

This procedure writes code process-specific source-code file (which need not be Fortran). It may be a no-op, if the source code is generated by Make instead.

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_code_file), deferred :: write_source_code

```

## Writer for Fortran-module matrix elements

If the matrix element is available as a Fortran module, we have specific requirements: (i) the features are imported via `use` directives, (ii) the specific features require `bind(C)` wrappers.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

```

<Prclib interfaces: public>+≡
  public :: prc_writer_f_module_t

<Prclib interfaces: types>+≡
  type, extends (prc_writer_t), abstract :: prc_writer_f_module_t
  contains
    <Prclib interfaces: prc writer f module: TBP>
  end type prc_writer_f_module_t

```

Return the name of the Fortran module. As a default implementation, we take the process ID unchanged.

```

<Prclib interfaces: prc writer f module: TBP>≡
  procedure, nopass :: get_module_name => prc_writer_get_module_name

<Prclib interfaces: procedures>+≡
  function prc_writer_get_module_name (id) result (name)
    type(string_t) :: name
    type(string_t), intent(in) :: id
    name = id
  end function prc_writer_get_module_name

```

Write a `use` directive that associates the driver reference with the procedure in the matrix element code. By default, we use the C name for this.

```

<Prclib interfaces: prc writer f module: TBP>+≡
  procedure :: write_use_line => prc_writer_write_use_line

<Prclib interfaces: procedures>+≡
  subroutine prc_writer_write_use_line (writer, unit, id, feature)
    class(prc_writer_f_module_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t) :: id, feature
    write (unit, "(2x,9A)") "use ", char (writer%get_module_name (id)), &
      ", only: ", char (writer%get_c_procname (id, feature)), &
      " => ", char (writer%get_procname (feature))
  end subroutine prc_writer_write_use_line

```

Write a wrapper routine for a feature. This also associates a C name the module procedure. The details depend on the writer variant.

```

<Prclib interfaces: prc writer f module: TBP>+≡
  procedure(prc_write_wrapper), deferred :: write_wrapper

<Prclib interfaces: interfaces>+≡
  abstract interface
    subroutine prc_write_wrapper (writer, unit, id, feature)
      import
      class(prc_writer_f_module_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
    end subroutine prc_write_wrapper
  end interface

```

This is used for testing only: initialize the writer with a specific MD5 sum string.

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure :: init_test => prc_writer_init_test

<Prclib interfaces: procedures>+≡
  subroutine prc_writer_init_test (writer)
    class(prc_writer_t), intent(out) :: writer
    writer%md5sum = "1234567890abcdef1234567890abcdef"
  end subroutine prc_writer_init_test

```

## Writer for C-library matrix elements

This applies if the matrix element is available as a C library or a Fortran library with `bind(C)` compatible interface. We can use the basic version.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

```

<Prclib interfaces: types>+≡
  type, extends (prc_writer_t), abstract :: prc_writer_c_lib_t
  contains
    <Prclib interfaces: prc writer c lib: TBP>
  end type prc_writer_c_lib_t

```

### 14.1.5 Process records in the library driver

A process record holds the process (component) ID, the physics `model_name`, and the array of `features` that are implemented by the corresponding matrix element code.

The `writer` component holds procedures. The procedures write source code for the current record, either for the driver or for the Makefile.

```

<Prclib interfaces: types>+≡
  type :: prclib_driver_record_t
    type(string_t) :: id
    type(string_t) :: model_name
    type(string_t), dimension(:), allocatable :: feature
    class(prc_writer_t), pointer :: writer => null ()
  contains
    <Prclib interfaces: prclib driver record: TBP>
  end type prclib_driver_record_t

```

Output routine. We indent the output, so it smoothly integrates into the output routine for the whole driver.

Note: the pointer `writer` is introduced as a workaround for a NAG compiler bug.

```

<Prclib interfaces: prclib driver record: TBP>≡
  procedure :: write => prclib_driver_record_write

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_record_write (object, unit)
    class(prclib_driver_record_t), intent(in) :: object
    integer, intent(in) :: unit
    integer :: j
    class(prc_writer_t), pointer :: writer
    write (unit, "(3x,A,2x,['',A,']')") &
      char (object%id), char (object%model_name)
    if (allocated (object%feature)) then
      writer => object%writer
      write (unit, "(5x,A,A)", advance="no") &
        char (writer%type_name ()), ":"
      do j = 1, size (object%feature)
        write (unit, "(1x,A)", advance="no") &
          char (object%feature(j))
      end do
      write (unit, *)
    end if
  end subroutine prclib_driver_record_write

```

Get the C procedure name for a feature.

```

<Prclib interfaces: prclib driver record: TBP>+≡
  procedure :: get_c_procname => prclib_driver_record_get_c_procname

<Prclib interfaces: procedures>+≡
  function prclib_driver_record_get_c_procname (record, feature) result (name)
    type(string_t) :: name
    class(prclib_driver_record_t), intent(in) :: record
    type(string_t), intent(in) :: feature

```

```

        name = record%writer%get_c_procname (record%id, feature)
    end function prclib_driver_record_get_c_procname

```

Write a USE directive for a given feature. Applies only if the record corresponds to a Fortran module.

```

<Prclib interfaces: prclib driver record: TBP>+≡
    procedure :: write_use_line => prclib_driver_record_write_use_line
<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_record_write_use_line (record, unit, feature)
        class(prclib_driver_record_t), intent(in) :: record
        integer, intent(in) :: unit
        type(string_t), intent(in) :: feature
        select type (writer => record%writer)
            class is (prc_writer_f_module_t)
                call writer%write_use_line (unit, record%id, feature)
            end select
    end subroutine prclib_driver_record_write_use_line

```

The alternative: write an interface block for a given feature, unless the record corresponds to a Fortran module.

```

<Prclib interfaces: prclib driver record: TBP>+≡
    procedure :: write_interface => prclib_driver_record_write_interface
<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_record_write_interface (record, unit, feature)
        class(prclib_driver_record_t), intent(in) :: record
        integer, intent(in) :: unit
        type(string_t), intent(in) :: feature
        select type (writer => record%writer)
            class is (prc_writer_f_module_t)
                class default
                    call writer%write_interface (unit, record%id, feature)
                end select
        end select
    end subroutine prclib_driver_record_write_interface

```

Write all special feature interfaces for the current record. Do this for all process variants.

```

<Prclib interfaces: prclib driver record: TBP>+≡
    procedure :: write_interfaces => prclib_driver_record_write_interfaces
<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_record_write_interfaces (record, unit)
        class(prclib_driver_record_t), intent(in) :: record
        integer, intent(in) :: unit
        integer :: i
        do i = 1, size (record%feature)
            call record%writer%write_interface (unit, record%id, record%feature(i))
        end do
    end subroutine prclib_driver_record_write_interfaces

```

Write the wrapper routines for this record, if it corresponds to a Fortran module.

```

<Prclib interfaces: prclib driver record: TBP>+≡
    procedure :: write_wrappers => prclib_driver_record_write_wrappers

```

```

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_record_write_wrappers (record, unit)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    integer :: i
    select type (writer => record%writer)
    class is (prc_writer_f_module_t)
      do i = 1, size (record%feature)
        call writer%write_wrapper (unit, record%id, record%feature(i))
      end do
    end select
  end subroutine prclib_driver_record_write_wrappers

```

Write the Makefile code for this record.

```

<Prclib interfaces: prclib driver record: TBP>+≡
  procedure :: write_makefile_code => prclib_driver_record_write_makefile_code

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_record_write_makefile_code (record, unit, os_data)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    type(os_data_t), intent(in) :: os_data
    call record%writer%write_makefile_code (unit, record%id, os_data)
  end subroutine prclib_driver_record_write_makefile_code

```

Write source-code files for this record. (Does nothing if the source code is handled by Makefile rules.)

```

<Prclib interfaces: prclib driver record: TBP>+≡
  procedure :: write_source_code => prclib_driver_record_write_source_code

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_record_write_source_code (record)
    class(prclib_driver_record_t), intent(in) :: record
    call record%writer%write_source_code (record%id)
  end subroutine prclib_driver_record_write_source_code

```

### 14.1.6 The process library driver object

A `prclib_driver_t` object provides the interface to external matrix element code. The code is provided by an external library which is either statically or dynamically linked.

The `basename` identifies the library, both by file names and by Fortran variable names.

The `static` flag tells whether the external library is statically or dynamically linked.

The `loaded` flag becomes true once all procedure pointers to the matrix element have been assigned.

For a dynamical external library, the communication proceeds via a `dlaccess` object.

`n_processes` is the number of external process code components that are referenced by this library. The code is addressed by index (`i_lib` in the process

library entry above). This number should be equal to the number returned by `get_n_prc`.

For each external process, there is a separate **record** which holds the data that are needed for the driver parts which are specific for a given process component. The actual pointers for the loaded library will be assigned elsewhere.

The remainder is a collection of procedure pointers, which can be assigned once all external code has been compiled and linked. The procedure pointers all take a process component code index as an argument. Most return information about the process component that should match the process definition. The `get_fptr` procedures return a function pointer, which is the actual means to compute matrix elements or retrieve associated data.

Finally, the `unload_hook` and `reload_hook` pointers allow for the insertion of additional code when a library is loaded.

```

(Prclib interfaces: public)+≡
    public :: prclib_driver_t

(Prclib interfaces: types)+≡
    type :: prclib_driver_t
        type(string_t) :: basename
        character(32) :: md5sum = ""
        logical :: static = .false.
        logical :: loaded = .false.
        type(string_t) :: libname
        type(dlaccess_t) :: dlaccess
        integer :: n_processes = 0
        type(prclib_driver_record_t), dimension(:), allocatable :: record
        procedure(prc_get_n_processes), nopass, pointer :: &
            get_n_processes => null ()
        procedure(prc_get_stringptr), nopass, pointer :: &
            get_process_id_ptr => null ()
        procedure(prc_get_stringptr), nopass, pointer :: &
            get_model_name_ptr => null ()
        procedure(prc_get_stringptr), nopass, pointer :: &
            get_md5sum_ptr => null ()
        procedure(prc_get_log), nopass, pointer :: &
            get_omp_status => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_in => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_out => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_flv => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_hel => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_col => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_cin => null ()
        procedure(prc_get_int), nopass, pointer :: get_n_cf => null ()
        procedure(prc_set_int_tab1), nopass, pointer :: &
            set_flv_state_ptr => null ()
        procedure(prc_set_int_tab1), nopass, pointer :: &
            set_hel_state_ptr => null ()
        procedure(prc_set_col_state), nopass, pointer :: &
            set_col_state_ptr => null ()
        procedure(prc_set_color_factors), nopass, pointer :: &
            set_color_factors_ptr => null ()
        procedure(prc_get_fptr), nopass, pointer :: get_fptr => null ()
!       procedure(prclib_unload_hook), nopass, pointer :: unload_hook => null ()

```



```

!      procedure(prclib_reload_hook), nopass, pointer :: reload_hook => null ()
contains
  <Prclib interfaces: prclib driver: TBP>
end type prclib_driver_t

```

Print just the metadata. Procedure pointers cannot be printed.

```

<Prclib interfaces: prclib driver: TBP>≡
  procedure :: write => prclib_driver_write

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_write (object, unit)
    class(prclib_driver_t), intent(in) :: object
    integer, intent(in) :: unit
    integer :: i
    write (unit, "(1x,A,A)") &
      "External matrix-element code library: ", char (object%basename)
    write (unit, "(3x,A,L1)") "static      = ", object%static
    write (unit, "(3x,A,L1)") "loaded      = ", object%loaded
    write (unit, "(3x,A,A,A)") "MD5 sum    = '", object%md5sum, "'"
    write (unit, *)
    call object%dlaccess%write (unit)
    write (unit, *)
    if (allocated (object%record)) then
      write (unit, "(1x,A)") "Matrix-element code entries:"
      do i = 1, object%n_processes
        call object%record(i)%write (unit)
      end do
    else
      write (unit, "(1x,A)") "Matrix-element code entries: [undefined]"
    end if
  end subroutine prclib_driver_write

```

Initialize the ID array and set `n_processes` accordingly.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: init => prclib_driver_init

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_init (driver, basename, n_processes, static)
    class(prclib_driver_t), intent(out) :: driver
    type(string_t), intent(in) :: basename
    integer, intent(in) :: n_processes
    logical, intent(in), optional :: static
    driver%basename = basename
    if (present (static)) driver%static = static
    driver%n_processes = n_processes
    allocate (driver%record (n_processes))
  end subroutine prclib_driver_init

```

Set the MD5 sum. This is separate because the MD5 sum may be known only after initialization.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: set_md5sum => prclib_driver_set_md5sum

```

```

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_set_md5sum (driver, md5sum)
    class(prclib_driver_t), intent(inout) :: driver
    character(32), intent(in) :: md5sum
    driver%md5sum = md5sum
  end subroutine prclib_driver_set_md5sum

```

Set the process record for a specific library entry. If the index is zero, we do nothing.

NOTE: Using allocate-on-assignment for `feature` triggers a bug in gfortran 4.6.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: set_record => prclib_driver_set_record

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_set_record (driver, i, &
    id, model_name, features, writer)
    class(prclib_driver_t), intent(inout) :: driver
    integer, intent(in) :: i
    type(string_t), intent(in) :: id
    type(string_t), intent(in) :: model_name
    type(string_t), dimension(:), intent(in) :: features
    class(prc_writer_t), intent(in), pointer :: writer
    if (i > 0) then
      associate (record => driver%record(i))
        record%id = id
        record%model_name = model_name
        allocate (record%feature (size (features)))
        record%feature = features
        record%writer => writer
      end associate
    end if
  end subroutine prclib_driver_set_record

```

Write all USE directives for a given feature, scanning the array of processes. Only Fortran-module processes count. Then, write interface blocks for the remaining processes.

The `implicit none` statement must go in-between.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_interfaces => prclib_driver_write_interfaces

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_write_interfaces (driver, unit, feature)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: feature
    integer :: i
    do i = 1, driver%n_processes
      call driver%record(i)%write_use_line (unit, feature)
    end do
    write (unit, "(2x,9A)") "implicit none"
    do i = 1, driver%n_processes
      call driver%record(i)%write_interface (unit, feature)
    end do
  end subroutine prclib_driver_write_interfaces

```

```

end do
end subroutine prclib_driver_write_interfaces

```

### 14.1.7 Write makefile

The makefile contains constant parts, parts that depend on the library name, and parts that depend on the specific processes and their types.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: generate_makefile => prclib_driver_generate_makefile

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_generate_makefile (driver, unit, os_data)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(os_data_t), intent(in) :: os_data
    integer :: i
    write (unit, "(A)")  "# WHIZARD: Makefile for process library '" &
      // char (driver%basename) // "'"
    write (unit, "(A)")  "# Automatically generated file, do not edit"
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Integrity check (don't modify the following line!)"
    write (unit, "(A)")  "MD5SUM = '" // driver%md5sum // "'"
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Library name"
    write (unit, "(A)")  "BASE = " // char (driver%basename)
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Compiler"
    write (unit, "(A)")  "FC = " // char (os_data%fc)
    write (unit, "(A)")  "CC = " // char (os_data%cc)
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Included libraries"
    write (unit, "(A)")  "FCINCL = " // char (os_data%whizard_includes)
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Compiler flags"
    write (unit, "(A)")  "FCFLAGS = " // char (os_data%fcflags)
    write (unit, "(A)")  "FCFLAGS_PIC = " // char (os_data%fcflags_pic)
    write (unit, "(A)")  "CFLAGS = " // char (os_data%cflags)
    write (unit, "(A)")  "CFLAGS_PIC = " // char (os_data%cflags_pic)
    write (unit, "(A)")  "LDFLAGS = " // char (os_data%whizard_ldflags) &
      // " " // char (os_data%ldflags)
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Libtool"
    write (unit, "(A)")  "LIBTOOL = " // char (os_data%whizard_libtool)
    write (unit, "(A)")  "FCOMPILE = $(LIBTOOL) --tag=FC --mode=compile"
    write (unit, "(A)")  "CCOMPILE = $(LIBTOOL) --tag=CC --mode=compile"
    write (unit, "(A)")  "LINK = $(LIBTOOL) --tag=FC --mode=link"
    write (unit, "(A)")  ""
    write (unit, "(A)")  "# Compile commands (default)"
    write (unit, "(A)")  "LTF_COMPILE = $(FCOMPILE) $(FC) -c &
      &$(FCINCL) $(FCFLAGS) $(FCFLAGS_PIC)"
    write (unit, "(A)")  "LTCC_COMPILE = $(CCOMPILE) $(CC) -c &
      &$(CFLAGS) $(CFLAGS_PIC)"
    write (unit, "(A)")  ""

```

```

write (unit, "(A)")  "# Default target"
write (unit, "(A)")  "all: link"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Matrix-element code files"
do i = 1, size (driver%record)
    call driver%record(i)%write_makefile_code (unit, os_data)
end do
write (unit, "(A)")  ""
write (unit, "(A)")  "# Library driver"
write (unit, "(A)")  "$(BASE).lo: $(BASE).f90 $(OBJECTS)"
write (unit, "(A)")  TAB // "$(LTF_COMPILE) $<"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Library"
write (unit, "(A)")  "$(BASE).la: $(BASE).lo $(OBJECTS)"
write (unit, "(A)")  TAB // "$(LINK) $(FC) -module -rpath /dev/null &
    &$(FCFLAGS) $(LD_FLAGS) -o $(BASE).la $~"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Main targets"
write (unit, "(A)")  "link: compile $(BASE).la"
write (unit, "(A)")  "compile: source $(OBJECTS) $(BASE).lo"
write (unit, "(A)")  "source: $(SOURCES) $(BASE).f90"
write (unit, "(A)")  ".PHONY: link compile source"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Specific cleanup targets"
do i = 1, size (driver%record)
    write (unit, "(A)")  "clean-" // char (driver%record(i)%id) // ":"
    write (unit, "(A)")  ".PHONY: clean-" // char (driver%record(i)%id)
end do
write (unit, "(A)")  ""
write (unit, "(A)")  "# Generic cleanup targets"
write (unit, "(A)")  "clean-library:"
write (unit, "(A)")  TAB // "rm -f $(BASE).la"
write (unit, "(A)")  "clean-objects:"
write (unit, "(A)")  TAB // "rm -f $(BASE).lo $(CLEAN_OBJECTS)"
write (unit, "(A)")  "clean-source:"
write (unit, "(A)")  TAB // "rm -f $(CLEAN_SOURCES)"
write (unit, "(A)")  "clean-driver:"
write (unit, "(A)")  TAB // "rm -f $(BASE).f90"
write (unit, "(A)")  "clean-makefile:"
write (unit, "(A)")  TAB // "rm -f $(BASE).makefile"
write (unit, "(A)")  ".PHONY: clean-library clean-objects &
    &clean-source clean-driver clean-makefile"
write (unit, "(A)")  ""
write (unit, "(A)")  "clean: clean-library clean-objects clean-source"
write (unit, "(A)")  "distclean: clean clean-driver clean-makefile"
write (unit, "(A)")  ".PHONY: clean distclean"
end subroutine prclib_driver_generate_makefile

```

### 14.1.8 Write driver file

This procedure writes the process library driver source code to the specified output unit. The individual routines for writing source-code procedures are

given below.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: generate_driver_code => prclib_driver_generate_code

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_generate_code (driver, unit)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t) :: prefix
    integer :: i

    prefix = driver%basename // "_"

    write (unit, "(A)")  "! WHIZARD matrix-element code interface"
    write (unit, "(A)")  "!"
    write (unit, "(A)")  "! Automatically generated file, do not edit"
    call driver%write_lib_md5sum_fun (unit, prefix)
    call driver%write_get_n_processes_fun (unit, prefix)
    call driver%write_get_process_id_fun (unit, prefix)
    call driver%write_get_model_name_fun (unit, prefix)
!   call write_get_restrictions_fun ()
!   call write_get_omega_flags_fun ()
    call driver%write_get_md5sum_fun (unit, prefix)
    call driver%write_string_to_array_fun (unit, prefix)
    call driver%write_get_openmp_status_fun (unit, prefix)
    call driver%write_get_int_fun (unit, prefix, var_str ("n_in"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_out"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_flv"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_hel"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_col"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_cin"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_cf"))
    call driver%write_set_int_sub (unit, prefix, var_str ("flv_state"))
    call driver%write_set_int_sub (unit, prefix, var_str ("hel_state"))
    call driver%write_set_col_state_sub (unit, prefix)
    call driver%write_set_color_factors_sub (unit, prefix)
    call driver%write_get_fptr_sub (unit, prefix)
    do i = 1, driver%n_processes
      call driver%record(i)%write_wrappers (unit)
    end do
  end subroutine prclib_driver_generate_code

```

This function provides the overall library MD5sum. The function is for internal use (therefore not bind(C)), the external interface is via the `get_md5sum_ptr` procedure with index 0.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_lib_md5sum_fun => prclib_driver_write_lib_md5sum_fun

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_write_lib_md5sum_fun (driver, unit, prefix)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(A)")  ""

```

```

write (unit, "(A)")  "! The MD5 sum of the library"
write (unit, "(A)")  "function " // char (prefix) &
// "md5sum () result (md5sum)"
write (unit, "(A)")  "  implicit none"
write (unit, "(A)")  "  character(32) :: md5sum"
write (unit, "(A)")  "  md5sum = ' " // driver%md5sum // "' "
write (unit, "(A)")  "end function " // char (prefix) // "md5sum"
end subroutine prclib_driver_write_lib_md5sum_fun

```

### 14.1.9 Interface bodies for informational functions

These interfaces implement the communication between WHIZARD (the main program) and the process-library driver. The procedures are all BIND(C), so they can safely be exposed by the library and handled by the `dlopen` mechanism, which apparently understands only C calling conventions.

In the sections below, for each procedure, we provide both the interface itself and a procedure that writes the corresponding procedure as source code to the process library driver.

#### Process count

Return the number of processes contained in the library.

```

<Prclib interfaces: public>+≡
  public :: prc_get_n_processes

<Prclib interfaces: interfaces>+≡
  abstract interface
    function prc_get_n_processes () result (n) bind(C)
    import
    integer(c_int) :: n
    end function prc_get_n_processes
  end interface

```

Here is the code.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_get_n_processes_fun

<Prclib interfaces: procedures>+≡
  subroutine write_get_n_processes_fun (driver, unit, prefix)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(A)")  ""
    write (unit, "(A)")  "! Return the number of processes in this library"
    write (unit, "(A)")  "function " // char (prefix) &
// "get_n_processes () result (n) bind(C)"
    write (unit, "(A)")  "  use iso_c_binding"
    write (unit, "(A)")  "  implicit none"
    write (unit, "(A)")  "  integer(c_int) :: n"
    write (unit, "(A,I0)")  "  n = ", driver%n_processes
    write (unit, "(A)")  "end function " // char (prefix) &
// "get_n_processes"
  end subroutine write_get_n_processes_fun

```

## Informational string functions

These functions return constant information about the matrix-element code.

The following procedures have to return strings. With the BIND(C) constraint, we choose to return the C pointer to a string, and its length, so the procedures implement this interface. They are actually subroutines.

```
(Prclib interfaces: public)+≡
    public :: prc_get_stringptr

(Prclib interfaces: interfaces)+≡
    abstract interface
        subroutine prc_get_stringptr (i, cptr, len) bind(C)
            import
            integer(c_int), intent(in) :: i
            type(c_ptr), intent(out) :: cptr
            integer(c_int), intent(out) :: len
        end subroutine prc_get_stringptr
    end interface
```

To hide this complication, we introduce a subroutine that converts the returned C pointer to a **string\_t** object. As a side effect, we deallocate the original after conversion – otherwise, we might have a memory leak.

For the conversion, we first pointer-convert the C pointer to a Fortran character array pointer, length 1 and size **len**. Using argument association and an internal subroutine, we convert this to a character array with length **len** and size 1. Using ordinary assignment, we finally convert this to **string\_t**.

The function takes the pointer-returning function as an argument. The index **i** identifies the process in the library.

```
(Prclib interfaces: procedures)+≡
    subroutine get_string_via_cptr (string, i, get_stringptr)
        type(string_t), intent(out) :: string
        integer, intent(in) :: i
        procedure(prc_get_stringptr) :: get_stringptr
        type(c_ptr) :: cptr
        integer(c_int) :: pid, len
        character(kind=c_char), dimension(:), pointer :: c_array
        pid = i
        call get_stringptr (pid, cptr, len)
        if (c_associated (cptr)) then
            call c_f_pointer (cptr, c_array, shape = [len])
            call set_string (c_array)
            call get_stringptr (0_c_int, cptr, len)
        else
            string = ""
        end if
    contains
        subroutine set_string (buffer)
            character(len, kind=c_char), dimension(1), intent(in) :: buffer
            string = buffer(1)
        end subroutine set_string
    end subroutine get_string_via_cptr
```

Since the module procedures return Fortran strings, we have to convert them. This is the necessary auxiliary routine. The routine is not BIND(C), it is not accessed from outside.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure, nopass :: write_string_to_array_fun

<Prclib interfaces: procedures>+≡
  subroutine write_string_to_array_fun (unit, prefix)
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(A)") ""
    write (unit, "(A)") "! Auxiliary: convert character string &
      &to array pointer"
    write (unit, "(A)") "subroutine " // char (prefix) &
      // "string_to_array (string, a)"
    write (unit, "(A)") "  use iso_c_binding"
    write (unit, "(A)") "  implicit none"
    write (unit, "(A)") "  character(*), intent(in) :: string"
    write (unit, "(A)") "  character(kind=c_char), dimension(:), &
      &allocatable, intent(out) :: a"
    write (unit, "(A)") "  integer :: i"
    write (unit, "(A)") "  allocate (a (len (string)))"
    write (unit, "(A)") "  do i = 1, size (a)"
    write (unit, "(A)") "    a(i) = string(i:i)"
    write (unit, "(A)") "  end do"
    write (unit, "(A)") "end subroutine " // char (prefix) &
      // "string_to_array"
  end subroutine write_string_to_array_fun

```

The above routine is called by other functions. It is not in a module, so they need its interface explicitly.

```

<Prclib interfaces: procedures>+≡
  subroutine write_string_to_array_interface (unit, prefix)
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(2x,A)") "interface"
    write (unit, "(2x,A)") "  subroutine " // char (prefix) &
      // "string_to_array (string, a)"
    write (unit, "(2x,A)") "    use iso_c_binding"
    write (unit, "(2x,A)") "    implicit none"
    write (unit, "(2x,A)") "    character(*), intent(in) :: string"
    write (unit, "(2x,A)") "    character(kind=c_char), dimension(:), &
      &allocatable, intent(out) :: a"
    write (unit, "(2x,A)") "  end subroutine " // char (prefix) &
      // "string_to_array"
    write (unit, "(2x,A)") "end interface"
  end subroutine write_string_to_array_interface

```

Here are the info functions which return strings, implementing the interface `prc_get_stringptr`.

Return the process ID for each process.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_get_process_id_fun

```



*<Prclib interfaces: procedures>+≡*

```

subroutine write_get_process_id_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)") ""
  write (unit, "(A)") "! Return the process ID of process #i &
    &(as a C pointer to a character array)"
  write (unit, "(A)") "subroutine " // char (prefix) &
    // "get_process_id_ptr (i, cptr, len) bind(C)"
  write (unit, "(A)") " use iso_c_binding"
  write (unit, "(A)") " implicit none"
  write (unit, "(A)") " integer(c_int), intent(in) :: i"
  write (unit, "(A)") " type(c_ptr), intent(inout) :: cptr"
  write (unit, "(A)") " integer(c_int), intent(out) :: len"
  write (unit, "(A)") " character(kind=c_char), dimension(:), &
    &allocatable, target, save :: a"
  call write_string_to_array_interface (unit, prefix)
  write (unit, "(A)") " select case (i)"
  write (unit, "(A)") " case (0); if (allocated (a)) deallocate (a)"
  do i = 1, driver%n_processes
    write (unit, "(A,I0,9A)") " case (" , i, ")"; " , &
      "call " , char (prefix), "string_to_array ('", &
      char (driver%record(i)%id), "'", a)"
  end do
  write (unit, "(A)") " end select"
  write (unit, "(A)") " if (allocated (a)) then"
  write (unit, "(A)") "   cptr = c_loc (a)"
  write (unit, "(A)") "   len = size (a)"
  write (unit, "(A)") " else"
  write (unit, "(A)") "   cptr = c_null_ptr"
  write (unit, "(A)") "   len = 0"
  write (unit, "(A)") " end if"
  write (unit, "(A)") "end subroutine " // char (prefix) &
    // "get_process_id_ptr"
end subroutine write_get_process_id_fun

```

Return the model name, given explicitly.

*<Prclib interfaces: prclib driver: TBP>+≡*

procedure :: write\_get\_model\_name\_fun

*<Prclib interfaces: procedures>+≡*

```

subroutine write_get_model_name_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)") ""
  write (unit, "(A)") "! Return the model name for process #i &
    &(as a C pointer to a character array)"
  write (unit, "(A)") "subroutine " // char (prefix) &
    // "get_model_name_ptr (i, cptr, len) bind(C)"
  write (unit, "(A)") " use iso_c_binding"

```

```

write (unit, "(A)") " implicit none"
write (unit, "(A)") " integer(c_int), intent(in) :: i"
write (unit, "(A)") " type(c_ptr), intent(inout) :: cptr"
write (unit, "(A)") " integer(c_int), intent(out) :: len"
write (unit, "(A)") " character(kind=c_char), dimension(:), &
&allocatable, target, save :: a"
call write_string_to_array_interface (unit, prefix)
write (unit, "(A)") " select case (i)"
write (unit, "(A)") " case (0); if (allocated (a)) deallocate (a)"
do i = 1, driver%n_processes
  write (unit, "(A,I0,9A)") " case (" , i, ")"; " , &
  "call " , char (prefix), "string_to_array ('" , &
  char (driver%record(i)%model_name), &
  "' , a)"
end do
write (unit, "(A)") " end select"
write (unit, "(A)") " if (allocated (a)) then"
write (unit, "(A)") "   cptr = c_loc (a)"
write (unit, "(A)") "   len = size (a)"
write (unit, "(A)") " else"
write (unit, "(A)") "   cptr = c_null_ptr"
write (unit, "(A)") "   len = 0"
write (unit, "(A)") " end if"
write (unit, "(A)") "end subroutine " // char (prefix) &
// "get_model_name_ptr"
end subroutine write_get_model_name_fun

```

Call the MD5 sum function for the process. The function calls the corresponding function of the matrix-element code, and it returns the C address of a character array with length 32.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_get_md5sum_fun

<Prclib interfaces: procedures>+≡
  subroutine write_get_md5sum_fun (driver, unit, prefix)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    integer :: i
    write (unit, "(A)") ""
    write (unit, "(A)") " ! Return the MD5 sum for the process configuration &
      &(as a C pointer to a character array)"
    write (unit, "(A)") "subroutine " // char (prefix) &
      // "get_md5sum_ptr (i, cptr, len) bind(C)"
    write (unit, "(A)") " use iso_c_binding"
    call driver%write_interfaces (unit, var_str ("md5sum"))
    write (unit, "(A)") " interface"
    write (unit, "(A)") "   function " // char (prefix) &
      // "md5sum () result (md5sum)"
    write (unit, "(A)") "     character(32) :: md5sum"
    write (unit, "(A)") "   end function " // char (prefix) // "md5sum"
    write (unit, "(A)") " end interface"
    write (unit, "(A)") " integer(c_int), intent(in) :: i"
    write (unit, "(A)") " type(c_ptr), intent(out) :: cptr"

```

```

write (unit, "(A)") " integer(c_int), intent(out) :: len"
write (unit, "(A)") " character(kind=c_char), dimension(32), &
&target, save :: md5sum"
write (unit, "(A)") " select case (i)"
write (unit, "(A)") " case (0)"
write (unit, "(A)") " call copy (" // char (prefix) // "md5sum ())"
write (unit, "(A)") " cptr = c_loc (md5sum)"
do i = 1, driver%n_processes
  write (unit, "(A,I0,A)") " case (" , i, ")"
  call driver%record(i)%write_md5sum_call (unit)
end do
write (unit, "(A)") " case default"
write (unit, "(A)") " cptr = c_null_ptr"
write (unit, "(A)") " end select"
write (unit, "(A)") " len = 32"
write (unit, "(A)") "contains"
write (unit, "(A)") " subroutine copy (md5sum_tmp)"
write (unit, "(A)") " character, dimension(32), intent(in) :: &
&md5sum_tmp"
write (unit, "(A)") " md5sum = md5sum_tmp"
write (unit, "(A)") " end subroutine copy"
write (unit, "(A)") "end subroutine " // char (prefix) &
// "get_md5sum_ptr"
end subroutine write_get_md5sum_fun

```

The actual call depends on the type of matrix element.

```

<Prclib interfaces: prclib driver record: TBP>+≡
  procedure :: write_md5sum_call => prclib_driver_record_write_md5sum_call

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_record_write_md5sum_call (record, unit)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    call record%writer%write_md5sum_call (unit, record%id)
  end subroutine prclib_driver_record_write_md5sum_call

```

The interface goes into the writer base type:

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_code), deferred :: write_md5sum_call

In the Fortran module case, we take a detour. The string returned by the
Fortran function is copied into a fixed-size array. The copy routine is an internal
subroutine of get_md5sum_ptr. We return the C address of the target array.

<Prclib interfaces: prc writer f module: TBP>+≡
  procedure :: write_md5sum_call => prc_writer_f_module_write_md5sum_call

<Prclib interfaces: procedures>+≡
  subroutine prc_writer_f_module_write_md5sum_call (writer, unit, id)
    class(prc_writer_f_module_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    write (unit, "(5x,9A)") "call copy (" , &
      char (writer%get_c_procname (id, var_str ("md5sum"))), " ()"
    write (unit, "(5x,9A)") "cptr = c_loc (md5sum)"

```

```
end subroutine prc_writer_f_module_write_md5sum_call
```

In the C library case, the library function returns a C pointer, which we can just copy.

```
<Prclib interfaces: prc writer c lib: TBP>≡
  procedure :: write_md5sum_call => prc_writer_c_lib_write_md5sum_call

<Prclib interfaces: procedures>+≡
  subroutine prc_writer_c_lib_write_md5sum_call (writer, unit, id)
    class(prc_writer_c_lib_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    write (unit, "(5x,9A)") &
      "cptr = ", &
      char (writer%get_c_procname (id, var_str ("get_md5sum"))), " ()"
  end subroutine prc_writer_c_lib_write_md5sum_call
```

### Actual references to the info functions

The string-valued info functions return C character arrays. For the API of the library driver, we provide convenience functions which (re)convert those arrays into `string_t` objects.

```
<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: get_process_id => prclib_driver_get_process_id
  procedure :: get_model_name => prclib_driver_get_model_name
  procedure :: get_md5sum => prclib_driver_get_md5sum

<Prclib interfaces: procedures>+≡
  function prclib_driver_get_process_id (driver, i) result (string)
    type(string_t) :: string
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    call get_string_via_cptr (string, i, driver%get_process_id_ptr)
  end function prclib_driver_get_process_id

  function prclib_driver_get_model_name (driver, i) result (string)
    type(string_t) :: string
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    call get_string_via_cptr (string, i, driver%get_model_name_ptr)
  end function prclib_driver_get_model_name

  function prclib_driver_get_md5sum (driver, i) result (string)
    type(string_t) :: string
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    call get_string_via_cptr (string, i, driver%get_md5sum_ptr)
  end function prclib_driver_get_md5sum
```

## Informational logical functions

When returning a logical value, we use the C boolean type, which may differ from Fortran.

```
<Prclib interfaces: public>+≡
    public :: prc_get_log

<Prclib interfaces: interfaces>+≡
    abstract interface
        function prc_get_log (pid) result (l) bind(C)
            import
            integer(c_int), intent(in) :: pid
            logical(c_bool) :: l
        end function prc_get_log
    end interface
```

Return a logical flag which tells whether OpenMP is supported for a specific process code.

```
<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: write_get_openmp_status_fun

<Prclib interfaces: procedures>+≡
    subroutine write_get_openmp_status_fun (driver, unit, prefix)
        class(prclib_driver_t), intent(in) :: driver
        integer, intent(in) :: unit
        type(string_t), intent(in) :: prefix
        integer :: i
        write (unit, "(A)") ""
        write (unit, "(A)") "! Return the OpenMP support status"
        write (unit, "(A)") "function " // char (prefix) &
            // "get_openmp_status (i) result (openmp_status) bind(C)"
        write (unit, "(A)") " use iso_c_binding"
        call driver%write_interfaces (unit, var_str ("openmp_supported"))
        write (unit, "(A)") " integer(c_int), intent(in) :: i"
        write (unit, "(A)") " logical(c_bool) :: openmp_status"
        write (unit, "(A)") " select case (i)"
        do i = 1, driver%n_processes
            write (unit, "(A,I0,9A)") " case (" , i, "); ", &
                "openmp_status = ", &
                char (driver%record(i)%get_c_procname &
                    (var_str ("openmp_supported"))), " ()"
        end do
        write (unit, "(A)") " end select"
        write (unit, "(A)") "end function " // char (prefix) &
            // "get_openmp_status"
    end subroutine write_get_openmp_status_fun
```

## Informational integer functions

Various process metadata are integer values. We can use a single interface for all of them.

```
<Prclib interfaces: public>+≡
    public :: prc_get_int
```

```

<Prclib interfaces: interfaces>+≡
abstract interface
    function prc_get_int (pid) result (n) bind(C)
    import
        integer(c_int), intent(in) :: pid
        integer(c_int) :: n
    end function prc_get_int
end interface

```

This function returns any data of type integer, for each process.

```

<Prclib interfaces: prclib driver: TBP>+≡
procedure :: write_get_int_fun

<Prclib interfaces: procedures>+≡
subroutine write_get_int_fun (driver, unit, prefix, feature)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    type(string_t), intent(in) :: feature
    integer :: i
    write (unit, "(A)") ""
    write (unit, "(9A)") "! Return the value of ", char (feature)
    write (unit, "(9A)") "function ", char (prefix), &
        "get_", char (feature), " (pid)", &
        " result (", char (feature), ") bind(C)"
    write (unit, "(9A)") " use iso_c_binding"
    call driver%write_interfaces (unit, feature)
    write (unit, "(9A)") " integer(c_int), intent(in) :: pid"
    write (unit, "(9A)") " integer(c_int) :: ", char (feature)
    write (unit, "(9A)") " select case (pid)"
    do i = 1, driver%n_processes
        write (unit, "(2x,A,I0,9A)") "case (", i, "); ", &
            char (feature), " = ", &
            char (driver%record(i)%get_c_procname (feature)), &
            " ()"
    end do
    write (unit, "(9A)") " end select"
    write (unit, "(9A)") "end function ", char (prefix), &
        "get_", char (feature)
end subroutine write_get_int_fun

```

Write a case line that assigns the value of the external function to the current return value.

```

<Prclib interfaces: procedures>+≡
subroutine write_case_int_fun (record, unit, i, feature)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    integer, intent(in) :: i
    type(string_t), intent(in) :: feature
    write (unit, "(5x,A,I0,9A)") "case (", i, "); ", &
        char (feature), " = ", char (record%get_c_procname (feature))
end subroutine write_case_int_fun

```

## Flavor and helicity tables

Transferring tables is more complicated. First, a two-dimensional array.

```
<Prclib interfaces: public>+≡
    public :: prc_set_int_tab1

<Prclib interfaces: interfaces>+≡
    abstract interface
        subroutine prc_set_int_tab1 (pid, cptr, shape) bind(C)
            import
                integer(c_int), intent(in) :: pid
                integer(c_int), dimension(*), intent(in) :: cptr
                integer(c_int), dimension(2), intent(in) :: shape
            end subroutine prc_set_int_tab1
        end interface
```

This subroutine returns a table of integers.

```
<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: write_set_int_sub

<Prclib interfaces: procedures>+≡
    subroutine write_set_int_sub (driver, unit, prefix, feature)
        class(prclib_driver_t), intent(in) :: driver
        integer, intent(in) :: unit
        type(string_t), intent(in) :: prefix
        type(string_t), intent(in) :: feature
        integer :: i
        write (unit, "(A)") ""
        write (unit, "(9A)") "! Set table: ", char (feature)
        write (unit, "(9A)") "subroutine ", char (prefix), &
            "set_", char (feature), "_ptr (pid, ", char (feature), &
            ", shape) bind(C)"
        write (unit, "(9A)") " use iso_c_binding"
        call driver%write_interfaces (unit, feature)
        write (unit, "(9A)") " integer(c_int), intent(in) :: pid"
        write (unit, "(9A)") " integer(c_int), dimension(*), intent(out) :: ", &
            char (feature)
        write (unit, "(9A)") " integer(c_int), dimension(2), intent(in) :: shape"
        write (unit, "(9A)") " integer, dimension(:,,:), allocatable :: ", &
            char (feature), "_tmp"
        write (unit, "(9A)") " integer :: i, j"
        write (unit, "(9A)") " select case (pid)"
        do i = 1, driver%n_processes
            write (unit, "(2x,A,IO,A)") "case (", i, ")"
            call driver%record(i)%write_int_sub_call (unit, feature)
        end do
        write (unit, "(9A)") " end select"
        write (unit, "(9A)") "end subroutine ", char (prefix), &
            "set_", char (feature), "_ptr"
    end subroutine write_set_int_sub
```

The actual call depends on the type of matrix element.

```
<Prclib interfaces: prclib driver record: TBP>+≡
    procedure :: write_int_sub_call => prclib_driver_record_write_int_sub_call
```

```

<Prclib interfaces: procedures>+≡
subroutine prclib_driver_record_write_int_sub_call (record, unit, feature)
  class(prclib_driver_record_t), intent(in) :: record
  integer, intent(in) :: unit
  type(string_t), intent(in) :: feature
  call record%writer%write_int_sub_call (unit, record%id, feature)
end subroutine prclib_driver_record_write_int_sub_call

```

The interface goes into the writer base type:

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_feature_code), deferred :: write_int_sub_call

```

In the Fortran module case, we need an extra copy in the (academical) situation where default integer and `c_int` differ. Otherwise, we just associate a Fortran array with the C pointer and let the matrix-element subroutine fill the array.

```

<Prclib interfaces: prc writer f module: TBP>+≡
  procedure :: write_int_sub_call => prc_writer_f_module_write_int_sub_call

<Prclib interfaces: procedures>+≡
subroutine prc_writer_f_module_write_int_sub_call (writer, unit, id, feature)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(5x,9A)") "allocate (", char (feature), "_tmp ", &
    "(shape(1), shape(2)))"
  write (unit, "(5x,9A)") "call ", &
    char (writer%get_c_procname (id, feature)), &
    " (", char (feature), "_tmp)"
  write (unit, "(5x,9A)") "forall (i=1:shape(1), j=1:shape(2)) "
  write (unit, "(8x,9A)") char (feature), "(i + shape(1)*(j-1)) = ", &
    char (feature), "_tmp", "(i,j)"
  write (unit, "(5x,9A)") "end forall"
end subroutine prc_writer_f_module_write_int_sub_call

```

In the C library case, we just transfer the C pointer to the library function.

```

<Prclib interfaces: prc writer c lib: TBP>+≡
  procedure :: write_int_sub_call => prc_writer_c_lib_write_int_sub_call

<Prclib interfaces: procedures>+≡
subroutine prc_writer_c_lib_write_int_sub_call (writer, unit, id, feature)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(5x,9A)") "call ", &
    char (writer%get_c_procname (id, feature)), " (", char (feature), ")"
end subroutine prc_writer_c_lib_write_int_sub_call

```

## Color state table

The color-state specification needs a table of integers (one array per color flow) and a corresponding array of color-ghost flags.

```

<Prclib interfaces: public>+≡
  public :: prc_set_col_state

```



```

<Prclib interfaces: interfaces>+≡
abstract interface
  subroutine prc_set_col_state (pid, col_state, ghost_flag, shape) bind(C)
  import
    integer(c_int), intent(in) :: pid
    integer(c_int), dimension(*), intent(out) :: col_state
    logical(c_bool), dimension(*), intent(out) :: ghost_flag
    integer(c_int), dimension(3), intent(in) :: shape
  end subroutine prc_set_col_state
end interface

```

```

<Prclib interfaces: prclib driver: TBP>+≡
procedure :: write_set_col_state_sub

```

```

<Prclib interfaces: procedures>+≡
subroutine write_set_col_state_sub (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  type(string_t) :: feature
  feature = "col_state"
  write (unit, "(A)") ""
  write (unit, "(9A)") "! Set tables: col_state, ghost_flag"
  write (unit, "(9A)") "subroutine ", char (prefix), &
    "set_col_state_ptr (pid, col_state, ghost_flag, shape) bind(C)"
  write (unit, "(9A)") " use iso_c_binding"
  call driver%write_interfaces (unit, feature)
  write (unit, "(9A)") " integer(c_int), intent(in) :: pid"
  write (unit, "(9A)") &
    " integer(c_int), dimension(*), intent(out) :: col_state"
  write (unit, "(9A)") &
    " logical(c_bool), dimension(*), intent(out) :: ghost_flag"
  write (unit, "(9A)") &
    " integer(c_int), dimension(3), intent(in) :: shape"
  write (unit, "(9A)") &
    " integer, dimension(:,:), allocatable :: col_state_tmp"
  write (unit, "(9A)") &
    " logical, dimension(:,:), allocatable :: ghost_flag_tmp"
  write (unit, "(9A)") " integer :: i, j, k"
  write (unit, "(A)") " select case (pid)"
  do i = 1, driver%n_processes
    write (unit, "(A,I0,A)") " case (" , i, ")"
    call driver%record(i)%write_col_state_call (unit)
  end do
  write (unit, "(A)") " end select"
  write (unit, "(9A)") "end subroutine ", char (prefix), &
    "set_col_state_ptr"
end subroutine write_set_col_state_sub

```

The actual call depends on the type of matrix element.

```

<Prclib interfaces: prclib driver record: TBP>+≡
procedure :: write_col_state_call => prclib_driver_record_write_col_state_call

```

```

<Prclib interfaces: procedures>+≡
subroutine prclib_driver_record_write_col_state_call (record, unit)
  class(prclib_driver_record_t), intent(in) :: record
  integer, intent(in) :: unit
  call record%writer%write_col_state_call (unit, record%id)
end subroutine prclib_driver_record_write_col_state_call

```

The interface goes into the writer base type:

```

<Prclib interfaces: prc writer: TBP>+≡
  procedure(write_code), deferred :: write_col_state_call

```

In the Fortran module case, we need an extra copy in the (academical) situation where default integer and `c_int` differ. Otherwise, we just associate a Fortran array with the C pointer and let the matrix-element subroutine fill the array.

```

<Prclib interfaces: prc writer f module: TBP>+≡
  procedure :: write_col_state_call => prc_writer_f_module_write_col_state_call

<Prclib interfaces: procedures>+≡
subroutine prc_writer_f_module_write_col_state_call (writer, unit, id)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(9A)") "  allocate (col_state_tmp ", &
    "(shape(1), shape(2), shape(3)))"
  write (unit, "(5x,9A)") "allocate (ghost_flag_tmp ", &
    "(shape(2), shape(3)))"
  write (unit, "(5x,9A)") "call ", &
    char (writer%get_c_procname (id, var_str ("col_state"))), &
    " (col_state_tmp, ghost_flag_tmp)"
  write (unit, "(5x,9A)") "forall (i = 1:shape(2), j = 1:shape(3))"
  write (unit, "(8x,9A)") "forall (k = 1:shape(1))"
  write (unit, "(11x,9A)") &
    "col_state(k + shape(1) * (i + shape(2)*(j-1) - 1)) ", &
    "= col_state_tmp(k,i,j)"
  write (unit, "(8x,9A)") "end forall"
  write (unit, "(8x,9A)") &
    "ghost_flag(i + shape(2)*(j-1)) = ghost_flag_tmp(i,j)"
  write (unit, "(5x,9A)") "end forall"
end subroutine prc_writer_f_module_write_col_state_call

```

In the C library case, we just transfer the C pointer to the library function.

```

<Prclib interfaces: prc writer c lib: TBP>+≡
  procedure :: write_col_state_call => prc_writer_c_lib_write_col_state_call

<Prclib interfaces: procedures>+≡
subroutine prc_writer_c_lib_write_col_state_call (writer, unit, id)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)") "call ", &
    char (writer%get_c_procname (id, var_str ("col_state"))), &
    " (col_state, ghost_flag)"
end subroutine prc_writer_c_lib_write_col_state_call

```

## Color factors

For the color-factor information, we return two integer arrays and a complex array.

```

<Prclib interfaces: public>+≡
    public :: prc_set_color_factors

<Prclib interfaces: interfaces>+≡
    abstract interface
        subroutine prc_set_color_factors &
            (pid, cf_index1, cf_index2, color_factors, shape) bind(C)
        import
        integer(c_int), intent(in) :: pid
        integer(c_int), dimension(*), intent(out) :: cf_index1, cf_index2
        complex(c_default_complex), dimension(*), intent(out) :: color_factors
        integer(c_int), dimension(1), intent(in) :: shape
    end subroutine prc_set_color_factors
end interface

```

This subroutine returns the color-flavor factor table.

```

<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: write_set_color_factors_sub

<Prclib interfaces: procedures>+≡
    subroutine write_set_color_factors_sub (driver, unit, prefix)
        class(prclib_driver_t), intent(in) :: driver
        integer, intent(in) :: unit
        type(string_t), intent(in) :: prefix
        integer :: i
        type(string_t) :: feature
        feature = "color_factors"
        write (unit, "(A)") ""
        write (unit, "(A)") "! Set tables: color factors"
        write (unit, "(9A)") "subroutine ", char (prefix), &
            "set_color_factors_ptr (pid, cf_index1, cf_index2, color_factors, ", &
            "shape) bind(C)"
        write (unit, "(A)") " use iso_c_binding"
        write (unit, "(A)") " use kinds"
        write (unit, "(A)") " use omega_color"
        call driver%write_interfaces (unit, feature)
        write (unit, "(A)") " integer(c_int), intent(in) :: pid"
        write (unit, "(A)") " integer(c_int), dimension(1), intent(in) :: shape"
        write (unit, "(A)") " integer(c_int), dimension(*), intent(out) :: &
            &cf_index1, cf_index2"
        write (unit, "(A)") " complex(c_default_complex), dimension(*), &
            &intent(out) :: color_factors"
        write (unit, "(A)") " type(omega_color_factor), dimension(:), &
            &allocatable :: cf"
        write (unit, "(A)") " select case (pid)"
        do i = 1, driver%n_processes
            write (unit, "(2x,A,I0,A)") "case (", i, ")"
            call driver%record(i)%write_color_factors_call (unit)
        end do
        write (unit, "(A)") " end select"
        write (unit, "(A)") "end subroutine " // char (prefix) &

```

```

        // "set_color_factors_ptr"
    end subroutine write_set_color_factors_sub

```

The actual call depends on the type of matrix element.

```

<Prclib interfaces: prclib_driver_record: TBP>+≡
    procedure :: write_color_factors_call => prclib_driver_record_write_color_factors_call

<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_record_write_color_factors_call (record, unit)
        class(prclib_driver_record_t), intent(in) :: record
        integer, intent(in) :: unit
        call record%writer%write_color_factors_call (unit, record%id)
    end subroutine prclib_driver_record_write_color_factors_call

```

The interface goes into the writer base type:

```

<Prclib interfaces: prc_writer: TBP>+≡
    procedure(write_code), deferred :: write_color_factors_call

```

In the Fortran module case, the matrix-element procedure fills an array of `omega_color_factor` elements. We distribute this array among two integer arrays and one complex-valued array, for which we have the C pointers.

```

<Prclib interfaces: prc_writer_f_module: TBP>+≡
    procedure :: write_color_factors_call => prc_writer_f_module_write_color_factors_call

<Prclib interfaces: procedures>+≡
    subroutine prc_writer_f_module_write_color_factors_call (writer, unit, id)
        class(prc_writer_f_module_t), intent(in) :: writer
        integer, intent(in) :: unit
        type(string_t), intent(in) :: id
        write (unit, "(5x,A)") "allocate (cf (shape(1)))"
        write (unit, "(5x,9A)") "call ", &
            char (writer%get_c_procname (id, var_str ("color_factors"))), " (cf)"
        write (unit, "(5x,9A)") "cf_index1(1:shape(1)) = cf%i1"
        write (unit, "(5x,9A)") "cf_index2(1:shape(1)) = cf%i2"
        write (unit, "(5x,9A)") "color_factors(1:shape(1)) = cf%factor"
    end subroutine prc_writer_f_module_write_color_factors_call

```

In the C library case, we just transfer the C pointers to the library function. There are three arrays.

```

<Prclib interfaces: prc_writer_c_lib: TBP>+≡
    procedure :: write_color_factors_call => &
        prc_writer_c_lib_write_color_factors_call

<Prclib interfaces: procedures>+≡
    subroutine prc_writer_c_lib_write_color_factors_call (writer, unit, id)
        class(prc_writer_c_lib_t), intent(in) :: writer
        integer, intent(in) :: unit
        type(string_t), intent(in) :: id
        write (unit, "(5x,9A)") "call ", &
            char (writer%get_c_procname (id, var_str ("color_factors"))), &
            " (cf_index1, cf_index2, color_factors)"
    end subroutine prc_writer_c_lib_write_color_factors_call

```

### 14.1.10 Interfaces for C-library matrix element

If the matrix element code is not provided as a Fortran module but as a C or bind(C) Fortran library, we need explicit interfaces for the library functions. They are not identical to the Fortran module versions. They transfer pointers directly.

The implementation is part of the `prc_writer_c_lib` type, which serves as base type for all C-library writers. It writes specific interfaces depending on the feature.

We bind this as the method `write_standard_interface` instead of `write_interface`, because we have to override the latter. Otherwise we could not call the method because the writer type is abstract.

```
<Prclib interfaces: prc writer c lib: TBP>+=  
  procedure :: write_standard_interface => prc_writer_c_lib_write_interface  
  
<Prclib interfaces: procedures>+=  
  subroutine prc_writer_c_lib_write_interface (writer, unit, id, feature)  
    class(prc_writer_c_lib_t), intent(in) :: writer  
    integer, intent(in) :: unit  
    type(string_t), intent(in) :: id, feature  
    select case (char (feature))  
    case ("md5sum")  
      write (unit, "(2x,9A)") "interface"  
      write (unit, "(5x,9A)") "function ", &  
        char (writer%get_c_procname (id, var_str ("get_md5sum"))), &  
        " () result (c_ptr) bind(C)"  
      write (unit, "(7x,9A)") "import"  
      write (unit, "(7x,9A)") "implicit none"  
      write (unit, "(7x,9A)") "type(c_ptr) :: c_ptr"  
      write (unit, "(5x,9A)") "end function ", &  
        char (writer%get_c_procname (id, var_str ("get_md5sum")))  
      write (unit, "(2x,9A)") "end interface"  
    case ("openmp_supported")  
      write (unit, "(2x,9A)") "interface"  
      write (unit, "(5x,9A)") "function ", &  
        char (writer%get_c_procname (id, feature)), &  
        " () result (status) bind(C)"  
      write (unit, "(7x,9A)") "import"  
      write (unit, "(7x,9A)") "implicit none"  
      write (unit, "(7x,9A)") "logical(c_bool) :: status"  
      write (unit, "(5x,9A)") "end function ", &  
        char (writer%get_c_procname (id, feature))  
      write (unit, "(2x,9A)") "end interface"  
    case ("n_in", "n_out", "n_flv", "n_hel", "n_col", "n_cin", "n_cf")  
      write (unit, "(2x,9A)") "interface"  
      write (unit, "(5x,9A)") "function ", &  
        char (writer%get_c_procname (id, feature)), &  
        " () result (n) bind(C)"  
      write (unit, "(7x,9A)") "import"  
      write (unit, "(7x,9A)") "implicit none"  
      write (unit, "(7x,9A)") "integer(c_int) :: n"  
      write (unit, "(5x,9A)") "end function ", &  
        char (writer%get_c_procname (id, feature))  
      write (unit, "(2x,9A)") "end interface"
```

```

case ("flv_state", "hel_state")
  write (unit, "(2x,9A)") "interface"
  write (unit, "(5x,9A)") "subroutine ", &
    char (writer%get_c_procname (id, feature)), &
    " (", char (feature), ") bind(C)"
  write (unit, "(7x,9A)") "import"
  write (unit, "(7x,9A)") "implicit none"
  write (unit, "(7x,9A)") "integer(c_int), dimension(*), intent(out) ", &
    ":: ", char (feature)
  write (unit, "(5x,9A)") "end subroutine ", &
    char (writer%get_c_procname (id, feature))
  write (unit, "(2x,9A)") "end interface"
case ("col_state")
  write (unit, "(2x,9A)") "interface"
  write (unit, "(5x,9A)") "subroutine ", &
    char (writer%get_c_procname (id, feature)), &
    " (col_state, ghost_flag) bind(C)"
  write (unit, "(7x,9A)") "import"
  write (unit, "(7x,9A)") "implicit none"
  write (unit, "(7x,9A)") "integer(c_int), dimension(*), intent(out) ", &
    ":: col_state"
  write (unit, "(7x,9A)") "logical(c_bool), dimension(*), intent(out) ", &
    ":: ghost_flag"
  write (unit, "(5x,9A)") "end subroutine ", &
    char (writer%get_c_procname (id, feature))
  write (unit, "(2x,9A)") "end interface"
case ("color_factors")
  write (unit, "(2x,9A)") "interface"
  write (unit, "(5x,9A)") "subroutine ", &
    char (writer%get_c_procname (id, feature)), &
    " (cf_index1, cf_index2, color_factors) bind(C)"
  write (unit, "(7x,9A)") "import"
  write (unit, "(7x,9A)") "implicit none"
  write (unit, "(7x,9A)") "integer(c_int), dimension(*), &
    &intent(out) :: cf_index1"
  write (unit, "(7x,9A)") "integer(c_int), dimension(*), &
    &intent(out) :: cf_index2"
  write (unit, "(7x,9A)") "complex(c_default_complex), dimension(*), &
    &intent(out) :: color_factors"
  write (unit, "(5x,9A)") "end subroutine ", &
    char (writer%get_c_procname (id, feature))
  write (unit, "(2x,9A)") "end interface"
end select
end subroutine prc_writer_c_lib_write_interface

```

### 14.1.11 Retrieving the tables

In the previous section we had the writer routines for procedures that return tables, actually C pointers to tables. Here, we write convenience routines that unpack them and move the contents to suitable Fortran arrays.

The flavor and helicity tables are two-dimensional integer arrays. We use intermediate storage for correctly transforming C to Fortran data types.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: set_flv_state => prclib_driver_set_flv_state
  procedure :: set_hel_state => prclib_driver_set_hel_state

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_set_flv_state (driver, i, flv_state)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    integer, dimension(:,:), allocatable, intent(out) :: flv_state
    integer :: n_tot, n_flv
    integer(c_int) :: pid
    integer(c_int), dimension(:,:), allocatable :: c_flv_state
    pid = i
    n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
    n_flv = driver%get_n_flv (pid)
    allocate (flv_state (n_tot, n_flv))
    allocate (c_flv_state (n_tot, n_flv))
    call driver%set_flv_state_ptr &
      (pid, c_flv_state, int ([n_tot, n_flv], kind=c_int))
    flv_state = c_flv_state
  end subroutine prclib_driver_set_flv_state

  subroutine prclib_driver_set_hel_state (driver, i, hel_state)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    integer, dimension(:,:), allocatable, intent(out) :: hel_state
    integer :: n_tot, n_hel
    integer(c_int) :: pid
    integer(c_int), dimension(:,:), allocatable, target :: c_hel_state
    pid = i
    n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
    n_hel = driver%get_n_hel (pid)
    allocate (hel_state (n_tot, n_hel))
    allocate (c_hel_state (n_tot, n_hel))
    call driver%set_hel_state_ptr &
      (pid, c_hel_state, int ([n_tot, n_hel], kind=c_int))
    hel_state = c_hel_state
  end subroutine prclib_driver_set_hel_state

```

The color-flow table is three-dimensional, otherwise similar. We simultaneously set the ghost-flag table, which consists of logical entries.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: set_col_state => prclib_driver_set_col_state

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_set_col_state (driver, i, col_state, ghost_flag)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    integer, dimension(:,:,:), allocatable, intent(out) :: col_state
    logical, dimension(:,:), allocatable, intent(out) :: ghost_flag
    integer :: n_cin, n_tot, n_col
    integer(c_int) :: pid
    integer(c_int), dimension(:,:,:), allocatable :: c_col_state
    logical(c_bool), dimension(:,:), allocatable :: c_ghost_flag
    pid = i

```

```

n_cin = driver%get_n_cin (pid)
n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
n_col = driver%get_n_col (pid)
allocate (col_state (n_cin, n_tot, n_col))
allocate (c_col_state (n_cin, n_tot, n_col))
allocate (ghost_flag (n_tot, n_col))
allocate (c_ghost_flag (n_tot, n_col))
call driver%set_col_state_ptr (pid, &
    c_col_state, c_ghost_flag, int ([n_cin, n_tot, n_col], kind=c_int))
col_state = c_col_state
ghost_flag = c_ghost_flag
end subroutine prclib_driver_set_col_state

```

The color-factor table is a sparse matrix: a two-column array of indices and one array which contains the corresponding factors.

```

<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: set_color_factors => prclib_driver_set_color_factors

<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_set_color_factors (driver, i, color_factors, cf_index)
        class(prclib_driver_t), intent(in) :: driver
        integer, intent(in) :: i
        complex(default), dimension(:), allocatable, intent(out) :: color_factors
        integer, dimension(:,:), allocatable, intent(out) :: cf_index
        integer :: n_cf
        integer(c_int) :: pid
        complex(c_default_complex), dimension(:), allocatable, target :: c_color_factors
        integer(c_int), dimension(:), allocatable, target :: c_cf_index1
        integer(c_int), dimension(:), allocatable, target :: c_cf_index2
        pid = i
        n_cf = driver%get_n_cf (pid)
        allocate (color_factors (n_cf))
        allocate (c_color_factors (n_cf))
        allocate (c_cf_index1 (n_cf))
        allocate (c_cf_index2 (n_cf))
        call driver%set_color_factors_ptr (pid, &
            c_cf_index1, c_cf_index2, &
            c_color_factors, int ([n_cf], kind=c_int))
        color_factors = c_color_factors
        allocate (cf_index (2, n_cf))
        cf_index(1,:) = c_cf_index1
        cf_index(2,:) = c_cf_index2
    end subroutine prclib_driver_set_color_factors

```

### 14.1.12 Returning a procedure pointer

The functions that directly access the matrix element, event by event, are assigned to a process-specific driver object as procedure pointers. For the `dlopen` interface, we use C function pointers. This subroutine returns such a pointer:

```

<Prclib interfaces: public>+≡
    public :: prc_get_fptr

```



```

<Prclib interfaces: interfaces>+≡
  abstract interface
    subroutine prc_get_fptr (pid, fid, fptr) bind(C)
      import
        integer(c_int), intent(in) :: pid
        integer(c_int), intent(in) :: fid
        type(c_funptr), intent(out) :: fptr
    end subroutine prc_get_fptr
  end interface

```

This procedure writes the source code for the procedure pointer returning subroutine.

All C functions that are provided by the matrix element code of a specific process are handled here. The selection consists of a double layered **select case** construct.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: write_get_fptr_sub

<Prclib interfaces: procedures>+≡
  subroutine write_get_fptr_sub (driver, unit, prefix)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    integer :: i, j
    write (unit, "(A)") ""
    write (unit, "(A)") "! Return C pointer to a procedure:"
    write (unit, "(A)") "! pid = process index; fid = function index"
    write (unit, "(4A)") "subroutine ", char (prefix), "get_fptr ", &
      "(pid, fid, fptr) bind(C)"
    write (unit, "(A)") " use iso_c_binding"
    write (unit, "(A)") " use kinds"
    write (unit, "(A)") " implicit none"
    write (unit, "(A)") " integer(c_int), intent(in) :: pid"
    write (unit, "(A)") " integer(c_int), intent(in) :: fid"
    write (unit, "(A)") " type(c_funptr), intent(out) :: fptr"
    do i = 1, driver%n_processes
      call driver%record(i)%write_interfaces (unit)
    end do
    write (unit, "(A)") " select case (pid)"
    do i = 1, driver%n_processes
      write (unit, "(2x,A,IO,A)") "case (", i, ")"
      write (unit, "(5x,A)") "select case (fid)"
      associate (record => driver%record(i))
        do j = 1, size (record%feature)
          write (unit, "(5x,A,IO,9A)") "case (", j, "); ", &
            "fptr = c_funloc (", &
            char (record%get_c_procname (record%feature(j))), &
            ")"
        end do
      end associate
      write (unit, "(5x,A)") "end select"
    end do
    write (unit, "(A)") " end select"
    write (unit, "(3A)") "end subroutine ", char (prefix), "get_fptr"
  end subroutine write_get_fptr_sub

```

```
end subroutine write_get_fptr_sub
```

The procedures for which we want to return a pointer (the 'features' of the matrix element code) are actually Fortran module procedures. If we want to have a C signature, we must write wrapper functions for all of them. The procedures, their signatures, and the appropriate writer routines are specific for the process type.

To keep this generic, we do not provide the writer routines here, but just the interface for a writer routine. The actual routines are stored in the process record.

The **prefix** indicates the library, the **id** indicates the process, and **procname** is the bare name of the procedure to be written.

```
<Prclib interfaces: public>+≡
public :: write_driver_code

<Prclib interfaces: interfaces>+≡
abstract interface
  subroutine write_driver_code (unit, prefix, id, procname)
    import
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    type(string_t), intent(in) :: id
    type(string_t), intent(in) :: procname
  end subroutine write_driver_code
end interface
```

### 14.1.13 Hooks

Interface for additional library unload / reload hooks (currently unused!)

```
<Prclib interfaces: public>+≡
public :: prclib_unload_hook
public :: prclib_reload_hook

<Prclib interfaces: interfaces>+≡
abstract interface
  subroutine prclib_unload_hook (libname)
    import
    type(string_t), intent(in) :: libname
  end subroutine prclib_unload_hook

  subroutine prclib_reload_hook (libname)
    import
    type(string_t), intent(in) :: libname
  end subroutine prclib_reload_hook
end interface
```

### 14.1.14 Make source, compile, link

Since we should have written a Makefile, these tasks amount to simple **make** calls. Note that the Makefile targets depend on each other, so calling **link** executes also the **source** and **compile** steps, when necessary.

The first routine writes source-code files for the individual processes. First it calls the writer routines directly for each process, then it calls `make source`. The make command may either post-process the files, or it may do the complete work, e.g., calling an external program that generates the files.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: make_source => prclib_driver_make_source

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_make_source (driver, os_data)
    class(prclib_driver_t), intent(in) :: driver
    type(os_data_t), intent(in) :: os_data
    integer :: i
    do i = 1, driver%n_processes
      call driver%record(i)%write_source_code ()
    end do
    call os_system_call ("make source " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end subroutine prclib_driver_make_source

```

Compile matrix element source code and the driver source code.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: make_compile => prclib_driver_make_compile

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_make_compile (driver, os_data)
    class(prclib_driver_t), intent(in) :: driver
    type(os_data_t), intent(in) :: os_data
    call os_system_call ("make compile " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end subroutine prclib_driver_make_compile

```

Combine all matrix-element code together with the driver in a process library on disk.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: make_link => prclib_driver_make_link

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_make_link (driver, os_data)
    class(prclib_driver_t), intent(in) :: driver
    type(os_data_t), intent(in) :: os_data
    call os_system_call ("make link " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end subroutine prclib_driver_make_link

```

### 14.1.15 Clean up generated files

The task of cleaning any generated files should also be deferred to Makefile targets. Apart from removing everything, removing specific files may be useful for partial rebuilds. (Note that removing the makefile itself can only be done once, for obvious reasons.)

If there is no makefile, do nothing.

```

<Prclib interfaces: prclib driver: TBP>+≡

```

```

procedure :: clean_library => prclib_driver_clean_library
procedure :: clean_objects => prclib_driver_clean_objects
procedure :: clean_source => prclib_driver_clean_source
procedure :: clean_driver => prclib_driver_clean_driver
procedure :: clean_makefile => prclib_driver_clean_makefile
procedure :: clean => prclib_driver_clean
procedure :: distclean => prclib_driver_distclean

(Prclib interfaces: procedures)+≡
subroutine prclib_driver_clean_library (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
    call os_system_call ("make clean-library " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_library

subroutine prclib_driver_clean_objects (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
    call os_system_call ("make clean-objects " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_objects

subroutine prclib_driver_clean_source (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
    call os_system_call ("make clean-source " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_source

subroutine prclib_driver_clean_driver (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
    call os_system_call ("make clean-driver " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_driver

subroutine prclib_driver_clean_makefile (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
    call os_system_call ("make clean-makefile " // os_data%makeflags &
      // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_makefile

subroutine prclib_driver_clean (driver, os_data)

```

```

class(prclib_driver_t), intent(in) :: driver
type(os_data_t), intent(in) :: os_data
if (driver%makefile_exists ()) then
    call os_system_call ("make clean " // os_data%makeflags &
        // " -f " // driver%basename // ".makefile")
end if
end subroutine prclib_driver_clean

subroutine prclib_driver_distclean (driver, os_data)
class(prclib_driver_t), intent(in) :: driver
type(os_data_t), intent(in) :: os_data
if (driver%makefile_exists ()) then
    call os_system_call ("make distclean " // os_data%makeflags &
        // " -f " // driver%basename // ".makefile")
end if
end subroutine prclib_driver_distclean

```

This Make target should remove all files that apply to a specific process. We execute this when we want to force remaking source code. Note that source targets need not have prerequisites, so just calling `make_source` would not do anything if the files exist.

```

<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: clean_proc => prclib_driver_clean_proc

<Prclib interfaces: procedures>+≡
    subroutine prclib_driver_clean_proc (driver, i, os_data)
        class(prclib_driver_t), intent(in) :: driver
        integer, intent(in) :: i
        type(os_data_t), intent(in) :: os_data
        type(string_t) :: id
        if (driver%makefile_exists ()) then
            id = driver%record(i)%id
            call os_system_call ("make clean-" // driver%record(i)%id // " " &
                // os_data%makeflags &
                // " -f " // driver%basename // ".makefile")
        end if
    end subroutine prclib_driver_clean_proc

```

### 14.1.16 Further Tools

Check for the appropriate makefile.

```

<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: makefile_exists => prclib_driver_makefile_exists

<Prclib interfaces: procedures>+≡
    function prclib_driver_makefile_exists (driver) result (flag)
        class(prclib_driver_t), intent(in) :: driver
        logical :: flag
        inquire (file = char (driver%basename) // ".makefile", exist = flag)
    end function prclib_driver_makefile_exists

```

### 14.1.17 Load the library

Once the library has been linked, we can dlopen it and assign all procedure pointers to their proper places in the library driver object. The loaded flag is set only if all required pointers have become assigned.

```
<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: load => prclib_driver_load

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_load (driver, os_data, noerror)
    class(prclib_driver_t), intent(inout) :: driver
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: noerror
    type(c_funptr) :: c_fptr
    logical :: ignore

    ignore = .false.; if (present (noerror)) ignore = noerror

    driver%libname = os_get_dlname (driver%basename, os_data, noerror, noerror)
    if (driver%libname == "") return
    if (.not. dlaccess_is_open (driver%dlaccess)) then
      call dlaccess_init &
        (driver%dlaccess, var_str (.), driver%libname, os_data)
      if (.not. ignore) call driver%check_dlerror ()
    end if
    driver%loaded = dlaccess_is_open (driver%dlaccess)
    if (.not. driver%loaded) return

    c_fptr = driver%get_c_funptr (var_str ("get_n_processes"))
    call c_f_procpointer (c_fptr, driver%get_n_processes)
    driver%loaded = driver%loaded .and. associated (driver%get_n_processes)

    c_fptr = driver%get_c_funptr (var_str ("get_process_id_ptr"))
    call c_f_procpointer (c_fptr, driver%get_process_id_ptr)
    driver%loaded = driver%loaded .and. associated (driver%get_process_id_ptr)

    c_fptr = driver%get_c_funptr (var_str ("get_model_name_ptr"))
    call c_f_procpointer (c_fptr, driver%get_model_name_ptr)
    driver%loaded = driver%loaded .and. associated (driver%get_model_name_ptr)

    c_fptr = driver%get_c_funptr (var_str ("get_md5sum_ptr"))
    call c_f_procpointer (c_fptr, driver%get_md5sum_ptr)
    driver%loaded = driver%loaded .and. associated (driver%get_md5sum_ptr)

    c_fptr = driver%get_c_funptr (var_str ("get_openmp_status"))
    call c_f_procpointer (c_fptr, driver%get_openmp_status)
    driver%loaded = driver%loaded .and. associated (driver%get_openmp_status)

    c_fptr = driver%get_c_funptr (var_str ("get_n_in"))
    call c_f_procpointer (c_fptr, driver%get_n_in)
    driver%loaded = driver%loaded .and. associated (driver%get_n_in)

    c_fptr = driver%get_c_funptr (var_str ("get_n_out"))
    call c_f_procpointer (c_fptr, driver%get_n_out)
    driver%loaded = driver%loaded .and. associated (driver%get_n_out)
```

```

c_fptr = driver%get_c_funptr (var_str ("get_n_flv"))
call c_f_procpointer (c_fptr, driver%get_n_flv)
driver%loaded = driver%loaded .and. associated (driver%get_n_flv)

c_fptr = driver%get_c_funptr (var_str ("get_n_hel"))
call c_f_procpointer (c_fptr, driver%get_n_hel)
driver%loaded = driver%loaded .and. associated (driver%get_n_hel)

c_fptr = driver%get_c_funptr (var_str ("get_n_col"))
call c_f_procpointer (c_fptr, driver%get_n_col)
driver%loaded = driver%loaded .and. associated (driver%get_n_col)

c_fptr = driver%get_c_funptr (var_str ("get_n_cin"))
call c_f_procpointer (c_fptr, driver%get_n_cin)
driver%loaded = driver%loaded .and. associated (driver%get_n_cin)

c_fptr = driver%get_c_funptr (var_str ("get_n_cf"))
call c_f_procpointer (c_fptr, driver%get_n_cf)
driver%loaded = driver%loaded .and. associated (driver%get_n_cf)

c_fptr = driver%get_c_funptr (var_str ("set_flv_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_flv_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_flv_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_hel_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_hel_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_hel_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_col_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_col_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_col_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_color_factors_ptr"))
call c_f_procpointer (c_fptr, driver%set_color_factors_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_color_factors_ptr)

c_fptr = driver%get_c_funptr (var_str ("get_fptr"))
call c_f_procpointer (c_fptr, driver%get_fptr)
driver%loaded = driver%loaded .and. associated (driver%get_fptr)

end subroutine prclib_driver_load

```

Unload. To be sure, nullify the procedure pointers.

```

<Prclib interfaces: prclib_driver: TBP>+≡
  procedure :: unload => prclib_driver_unload

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_unload (driver)
    class(prclib_driver_t), intent(inout) :: driver
    if (dlaccess_is_open (driver%dlaccess)) then
      call dlaccess_final (driver%dlaccess)
      call driver%check_dlerror ()
    end if
  end subroutine

```

```

driver%loaded = .false.
nullify (driver%get_n_processes)
nullify (driver%get_process_id_ptr)
nullify (driver%get_model_name_ptr)
nullify (driver%get_md5sum_ptr)
nullify (driver%get_openmp_status)
nullify (driver%get_n_in)
nullify (driver%get_n_out)
nullify (driver%get_n_flv)
nullify (driver%get_n_hel)
nullify (driver%get_n_col)
nullify (driver%get_n_cin)
nullify (driver%get_n_cf)
nullify (driver%set_flv_state_ptr)
nullify (driver%set_hel_state_ptr)
nullify (driver%set_col_state_ptr)
nullify (driver%set_color_factors_ptr)
nullify (driver%get_fptr)
end subroutine prclib_driver_unload

```

This subroutine checks the dlerror content and issues a fatal error if it finds an error there.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: check_dlerror => prclib_driver_check_dlerror

<Prclib interfaces: procedures>+≡
  subroutine prclib_driver_check_dlerror (driver)
    class(prclib_driver_t), intent(in) :: driver
    if (dlaccess_has_error (driver%dlaccess)) then
      call msg_fatal (char (dlaccess_get_error (driver%dlaccess)))
    end if
  end subroutine prclib_driver_check_dlerror

```

Here we call the DL interface to retrieve the C pointer to a named procedure.

```

<Prclib interfaces: prclib driver: TBP>+≡
  procedure :: get_c_funptr => prclib_driver_get_c_funptr

<Prclib interfaces: procedures>+≡
  function prclib_driver_get_c_funptr (driver, feature) result (c_fptr)
    type(c_funptr) :: c_fptr
    class(prclib_driver_t), intent(inout) :: driver
    type(string_t), intent(in) :: feature
    type(string_t) :: prefix, full_name
    prefix = driver%basename // "_"
    full_name = prefix // feature
    if (driver%static) then
!      c_fptr = libmanager_get_c_funptr (char (prefix), char (fname))
    else
      c_fptr = dlaccess_get_c_funptr (driver%dlaccess, full_name)
      call driver%check_dlerror ()
    end if
  end function prclib_driver_get_c_funptr

```



### 14.1.18 MD5 sums

Recall the MD5 sum written in the Makefile

```
<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: get_md5sum_makefile => prclib_driver_get_md5sum_makefile

<Prclib interfaces: procedures>+≡
    function prclib_driver_get_md5sum_makefile (driver) result (md5sum)
        class(prclib_driver_t), intent(in) :: driver
        character(32) :: md5sum
        type(string_t) :: filename
        character(80) :: buffer, key, dummy
        logical :: exist
        integer :: u, iostat
        md5sum = ""
        filename = driver%basename // ".makefile"
        inquire (file = char (filename), exist = exist)
        if (exist) then
            u = free_unit ()
            open (u, file = char (filename), action = "read", status = "old")
            iostat = 0
            do
                read (u, "(A)", iostat = iostat) buffer
                if (iostat /= 0) exit
                buffer = adjustl (buffer)
                select case (buffer(1:9))
                    case ("MD5SUM = ")
                        read (buffer(11:), "(A32)") md5sum
                        exit
                end select
            end do
            close (u)
        end if
    end function prclib_driver_get_md5sum_makefile
```

Recall the MD5 sum written in the driver source code.

```
<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: get_md5sum_driver => prclib_driver_get_md5sum_driver

<Prclib interfaces: procedures>+≡
    function prclib_driver_get_md5sum_driver (driver) result (md5sum)
        class(prclib_driver_t), intent(in) :: driver
        character(32) :: md5sum
        type(string_t) :: filename
        character(80) :: buffer, key, dummy
        logical :: exist
        integer :: u, iostat
        md5sum = ""
        filename = driver%basename // ".f90"
        inquire (file = char (filename), exist = exist)
        if (exist) then
            u = free_unit ()
            open (u, file = char (filename), action = "read", status = "old")
            iostat = 0
            do
```

```

        read (u, "(A)", iostat = iostat)  buffer
        if (iostat /= 0)  exit
        buffer = adjustl (buffer)
        select case (buffer(1:9))
        case ("md5sum = ")
            read (buffer(11:), "(A32)")  md5sum
            exit
        end select
    end do
    close (u)
end if
end function prclib_driver_get_md5sum_driver

```

Recall the MD5 sum written in the matrix element source code.

```

<Prclib interfaces: prclib driver: TBP>+≡
    procedure :: get_md5sum_source => prclib_driver_get_md5sum_source

<Prclib interfaces: procedures>+≡
function prclib_driver_get_md5sum_source (driver, i) result (md5sum)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    character(32) :: md5sum
    type(string_t) :: filename
    character(80) :: buffer, key, dummy
    logical :: exist
    integer :: u, iostat
    md5sum = ""

    filename = driver%record(i)%id // ".f90"
    inquire (file = char (filename), exist = exist)
    if (exist) then
        u = free_unit ()
        open (u, file = char (filename), action = "read", status = "old")
        iostat = 0
        do
            read (u, "(A)", iostat = iostat)  buffer
            if (iostat /= 0)  exit
            buffer = adjustl (buffer)
            select case (buffer(1:9))
            case ("md5sum = ")
                read (buffer(11:), "(A32)")  md5sum
                exit
            end select
        end do
        close (u)
    end if
end function prclib_driver_get_md5sum_source

```

### 14.1.19 Test

This is the master for calling self-test procedures.

```

<Prclib interfaces: public>+≡
    public :: prclib_interfaces_test

```

```

<Prclib interfaces: tests>≡
  subroutine prclib_interfaces_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <Prclib interfaces: execute tests>
  end subroutine prclib_interfaces_test

```

## Empty process list

Test 1: Create a driver object and display its contents. One of the feature lists references a writer procedure; this is just a dummy that does nothing useful.

```

<Prclib interfaces: execute tests>≡
  call test (prclib_interfaces_1, "prclib_interfaces_1", &
    "create driver object", &
    u, results)

<Prclib interfaces: tests>+≡
  subroutine prclib_interfaces_1 (u)
    integer, intent(in) :: u
    type(prclib_driver_t) :: driver
    character(32), parameter :: md5sum = "prclib_interfaces_1_md5sum"
    class(prc_writer_t), pointer :: test_writer_1

    write (u, "(A)")  "* Test output: prclib_interfaces_1"
    write (u, "(A)")  "* Purpose: display the driver object contents"
    write (u, *)
    write (u, "(A)")  "* Create a prclib driver object"
    write (u, "(A)")

    call driver%init (var_str ("prclib"), 3)
    call driver%set_md5sum (md5sum)

    allocate (test_writer_1_t :: test_writer_1)

    call driver%set_record (1, var_str ("test1"), var_str ("test_model"), &
      [var_str ("init")], test_writer_1)

    call driver%set_record (2, var_str ("test2"), var_str ("foo_model"), &
      [var_str ("another_proc")], test_writer_1)

    call driver%set_record (3, var_str ("test3"), var_str ("test_model"), &
      [var_str ("init"), var_str ("some_proc")], test_writer_1)

    call driver%write (u)

    deallocate (test_writer_1)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prclib_interfaces_1"
  end subroutine prclib_interfaces_1

```

The writer: the procedures write just comment lines. We can fix an instance of this as a parameter (since it has no mutable content) and just reference the fixed parameter.

NOTE: temporarily made public.

```

(Prclib interfaces: test types)≡
type, extends (prc_writer_t) :: test_writer_1_t
contains
  procedure, nopass :: type_name => test_writer_1_type_name
  procedure :: write_makefile_code => test_writer_1_mk
  procedure :: write_source_code => test_writer_1_src
  procedure :: write_interface => test_writer_1_if
  procedure :: write_md5sum_call => test_writer_1_md5sum
  procedure :: write_int_sub_call => test_writer_1_int_sub
  procedure :: write_col_state_call => test_writer_1_col_state
  procedure :: write_color_factors_call => test_writer_1_col_factors
end type test_writer_1_t

(Prclib interfaces: tests)+≡
function test_writer_1_type_name () result (string)
  type(string_t) :: string
  string = "test_1"
end function test_writer_1_type_name

subroutine test_writer_1_mk (writer, unit, id, os_data)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  write (unit, "(5A)")  "# Makefile code for process ", char (id), &
    " goes here."
end subroutine test_writer_1_mk

subroutine test_writer_1_src (writer, id)
  class(test_writer_1_t), intent(in) :: writer
  type(string_t), intent(in) :: id
end subroutine test_writer_1_src

subroutine test_writer_1_if (writer, unit, id, feature)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(2x,9A)")  "! Interface code for ", &
    char (id), "_", char (writer%get_procname (feature)), &
    " goes here."
end subroutine test_writer_1_if

subroutine test_writer_1_md5sum (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "! MD5sum call for ", char (id), " goes here."
end subroutine test_writer_1_md5sum

```

```

subroutine test_writer_1_int_sub (writer, unit, id, feature)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(5x,9A)") "! ", char (feature), " call for ", &
    char (id), " goes here."
end subroutine test_writer_1_int_sub

subroutine test_writer_1_col_state (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)") "! col_state call for ", &
    char (id), " goes here."
end subroutine test_writer_1_col_state

subroutine test_writer_1_col_factors (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)") "! color_factors call for ", &
    char (id), " goes here."
end subroutine test_writer_1_col_factors

```

## Process library driver file

Test 2: Write the driver file for a test case with two processes. The first process needs no wrapper (C library), the second one needs wrappers (Fortran module library).

```

<Prclib interfaces: execute tests>+≡
  call test (prclib_interfaces_2, "prclib_interfaces_2", &
    "write driver file", &
    u, results)

<Prclib interfaces: tests>+≡
subroutine prclib_interfaces_2 (u)
  integer, intent(in) :: u
  type(prclib_driver_t) :: driver
  character(32), parameter :: md5sum = "prclib_interfaces_2_md5sum"
  class(prc_writer_t), pointer :: test_writer_1, test_writer_2

  write (u, "(A)")  "* Test output: prclib_interfaces_2"
  write (u, "(A)")  "* Purpose: check the generated driver source code"
  write (u, "(A)")
  write (u, "(A)")  "* Create a prclib driver object (2 processes)"
  write (u, "(A)")

  call driver%init (var_str ("prclib2"), 2)
  call driver%set_md5sum (md5sum)

  allocate (test_writer_1_t :: test_writer_1)
  allocate (test_writer_2_t :: test_writer_2)

```

```

call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
    [var_str ("proc1")], test_writer_1)

call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
    [var_str ("proc1"), var_str ("proc2")], test_writer_2)

call driver%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the driver file"
write (u, "(A)")  "* File contents:"
write (u, "(A)")

call driver%generate_driver_code (u)

deallocate (test_writer_1)
deallocate (test_writer_2)

write (u, "(A)")
write (u, "(A)")  "* Test output end: prclib_interfaces_2"
end subroutine prclib_interfaces_2

```

A writer with wrapper code: the procedures again write just comment lines. Since all procedures are NOPASS, we can reuse two of the TBP.

```

<Prclib interfaces: test types>+=
type, extends (prc_writer_f_module_t) :: test_writer_2_t
contains
    procedure, nopass :: type_name => test_writer_2_type_name
    procedure :: write_makefile_code => test_writer_2_mk
    procedure :: write_source_code => test_writer_2_src
    procedure :: write_interface => test_writer_2_if
    procedure :: write_wrapper => test_writer_2_wr
end type test_writer_2_t

<Prclib interfaces: tests>+=
function test_writer_2_type_name () result (string)
    type(string_t) :: string
    string = "test_2"
end function test_writer_2_type_name

subroutine test_writer_2_mk (writer, unit, id, os_data)
    class(test_writer_2_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    write (unit, "(5A)")  "# Makefile code for process ", char (id), &
        " goes here."
end subroutine test_writer_2_mk

subroutine test_writer_2_src (writer, id)
    class(test_writer_2_t), intent(in) :: writer
    type(string_t), intent(in) :: id
end subroutine test_writer_2_src

```

```

subroutine test_writer_2_if (writer, unit, id, feature)
  class(test_writer_2_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(2x,9A)")  "! Interface code for ", &
    char (writer%get_module_name (id)), "_", &
    char (writer%get_procname (feature)), " goes here."
end subroutine test_writer_2_if

subroutine test_writer_2_wr (writer, unit, id, feature)
  class(test_writer_2_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, *)
  write (unit, "(9A)")  "! Wrapper code for ", &
    char (writer%get_c_procname (id, feature)), " goes here."
end subroutine test_writer_2_wr

```

## Process library makefile

Test 3: Write the makefile for compiling and linking the process library (processes and driver code). There are two processes, one with one method, one with two methods.

To have predictable output, we reset the system-dependent initial components of `os_data` to known values.

```

<Prclib interfaces: execute tests>+≡
  call test (prclib_interfaces_3, "prclib_interfaces_3", &
    "write makefile", &
    u, results)

<Prclib interfaces: tests>+≡
subroutine prclib_interfaces_3 (u)
  integer, intent(in) :: u
  type(prclib_driver_t) :: driver
  type(os_data_t) :: os_data
  character(32), parameter :: md5sum = "prclib_interfaces_3_md5sum"
  class(prc_writer_t), pointer :: test_writer_1, test_writer_2

  call os_data_init (os_data)
  os_data%fc = "fortran-compiler"
  os_data%whizard_includes = "-I module-dir"
  os_data%fcflags = "-C=all"
  os_data%fcflags_pic = "-PIC"
  os_data%cc = "c-compiler"
  os_data%cflags = "-I include-dir"
  os_data%cflags_pic = "-PIC"
  os_data%whizard_ldflags = ""
  os_data%ldflags = ""
  os_data%whizard_libtool = "my-libtool"

  write (u, "(A)")  "* Test output: prclib_interfaces_3"
  write (u, "(A)")  "* Purpose: check the generated Makefile"

```

```

write (u, *)
write (u, "(A)")  "* Create a prclib driver object (2 processes)"
write (u, "(A)")

call driver%init (var_str ("prclib3"), 2)
call driver%set_md5sum (md5sum)

allocate (test_writer_1_t :: test_writer_1)
allocate (test_writer_2_t :: test_writer_2)

call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
    [var_str ("proc1")], test_writer_1)

call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
    [var_str ("proc1"), var_str ("proc2")], test_writer_2)

call driver%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write Makefile"
write (u, "(A)")  "* File contents:"
write (u, "(A)")

call driver%generate_makefile (u, os_data)

deallocate (test_writer_1)
deallocate (test_writer_2)

write (u, "(A)")
write (u, "(A)")  "* Test output end: prclib_interfaces_3"
end subroutine prclib_interfaces_3

```

## Compile test with Fortran module

Test 4: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran module, therefore we need a wrapper for the featured procedure.

```

<Prclib interfaces: execute tests>+≡
call test (prclib_interfaces_4, "prclib_interfaces_4", &
    "compile and link (Fortran module)", &
    u, results)

<Prclib interfaces: tests>+≡
subroutine prclib_interfaces_4 (u)
    integer, intent(in) :: u
    type(prclib_driver_t) :: driver
    class(prc_writer_t), pointer :: test_writer_4
    type(os_data_t) :: os_data
    integer :: u_file

    integer, dimension(:, :), allocatable :: flv_state
    integer, dimension(:, :), allocatable :: hel_state

```



```

integer, dimension(:,:,:), allocatable :: col_state
logical, dimension(:,:), allocatable :: ghost_flag
integer, dimension(:,:), allocatable :: cf_index
complex(default), dimension(:), allocatable :: color_factors
character(32), parameter :: md5sum = "prclib_interfaces_4_md5sum      "
character(32) :: md5sum_file

type(c_funptr) :: proc1_ptr
interface
  subroutine proc1_t (n) bind(C)
    import
    integer(c_int), intent(out) :: n
  end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_4"
write (u, "(A)")  "* Purpose: compile, link, and load process library"
write (u, "(A)")  "*           with (fake) matrix-element code &
               &as a Fortran module"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)

allocate (test_writer_4_t :: test_writer_4)
call test_writer_4%init_test ()

call driver%init (var_str ("prclib4"), 1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test4"), var_str ("Test_model"), &
  [var_str ("proc1")], test_writer_4)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write Makefile"
u_file = free_unit ()
open (u_file, file="prclib4.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from Makefile"
write (u, "(A)")

md5sum_file = driver%get_md5sum_makefile ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"

```

```

u_file = free_unit ()
open (u_file, file="prclib4.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from driver source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_driver ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from matrix-element source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_source (1)
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
  write (u, "(A)")  "* Call library functions:"
  write (u, *)
  write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
  write (u, "(1x,A,A,A)")  "process_id   = '", &
    char (driver%get_process_id (1)), "'"
  write (u, "(1x,A,A,A)")  "model_name   = '", &
    char (driver%get_model_name (1)), "'"
  write (u, "(1x,A,A,A)")  "md5sum (lib) = '", &
    char (driver%get_md5sum (0)), "'"
  write (u, "(1x,A,A,A)")  "md5sum (proc) = '", &
    char (driver%get_md5sum (1)), "'"
  write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
  write (u, "(1x,A,I0)")  "n_in   = ", driver%get_n_in (1)
  write (u, "(1x,A,I0)")  "n_out  = ", driver%get_n_out (1)
  write (u, "(1x,A,I0)")  "n_flv  = ", driver%get_n_flv (1)

```

```

write (u, "(1x,A,I0)") "n_hel = ", driver%get_n_hel (1)
write (u, "(1x,A,I0)") "n_col = ", driver%get_n_col (1)
write (u, "(1x,A,I0)") "n_cin = ", driver%get_n_cin (1)
write (u, "(1x,A,I0)") "n_cf = ", driver%get_n_cf (1)

call driver%set_flv_state (1, flv_state)
write (u, "(1x,A,10(1x,I0))") "flv_state =", flv_state

call driver%set_hel_state (1, hel_state)
write (u, "(1x,A,10(1x,I0))") "hel_state =", hel_state

call driver%set_col_state (1, col_state, ghost_flag)
write (u, "(1x,A,10(1x,I0))") "col_state =", col_state
write (u, "(1x,A,10(1x,L1))") "ghost_flag =", ghost_flag

call driver%set_color_factors (1, color_factors, cf_index)
write (u, "(1x,A,10(1x,F5.3))") "color_factors =", color_factors
write (u, "(1x,A,10(1x,I0))") "cf_index =", cf_index

call driver%get_fptr (1, 1, proc1_ptr)
call c_f_procpointer (proc1_ptr, proc1)
if (associated (proc1)) then
    write (u, *)
    call proc1 (n)
    write (u, "(1x,A,I0)") "proc1(1) = ", n
end if

end if

deallocate (test_writer_4)

write (u, "(A)")
write (u, "(A)") "* Test output end: prclib_interfaces_4"
end subroutine prclib_interfaces_4

```

This version of test-code writer actually writes an interface and wrapper code.

The wrapped function is a no-parameter function with integer result.

The stored MD5 sum may be modified.

We will reuse this later, therefore public.

*<Prclib interfaces: public>+≡*

```
public :: test_writer_4_t
```

*<Prclib interfaces: test types>+≡*

```

type, extends (prc_writer_f_module_t) :: test_writer_4_t
contains
    procedure, nopass :: type_name => test_writer_4_type_name
    procedure :: write_makefile_code => test_writer_4_mk
    procedure :: write_source_code => test_writer_4_src
    procedure :: write_interface => test_writer_4_if
    procedure :: write_wrapper => test_writer_4_wr
end type test_writer_4_t

```

*<Prclib interfaces: tests>+≡*

```

function test_writer_4_type_name () result (string)
    type(string_t) :: string
    string = "test_4"
end function test_writer_4_type_name

subroutine test_writer_4_mk (writer, unit, id, os_data)
    class(test_writer_4_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    write (unit, "(5A)") "SOURCES += ", char (id), ".f90"
    write (unit, "(5A)") "OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)") "CLEAN_SOURCES += ", char (id), ".f90"
    write (unit, "(5A)") "CLEAN_OBJECTS += ", char (id), ".mod"
    write (unit, "(5A)") "CLEAN_OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)") char (id), ".lo: ", char (id), ".f90"
    write (unit, "(5A)") TAB, "$ (LTF_COMPILE) $<"
end subroutine test_writer_4_mk

subroutine test_writer_4_src (writer, id)
    class(test_writer_4_t), intent(in) :: writer
    type(string_t), intent(in) :: id
    call write_test_module_file (id, var_str ("proc1"), writer%md5sum)
end subroutine test_writer_4_src

subroutine test_writer_4_if (writer, unit, id, feature)
    class(test_writer_4_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
    write (unit, "(2x,9A)") "interface"
    write (unit, "(5x,9A)") "subroutine ", &
        char (writer%get_c_procname (id, feature)), &
        " (n) bind(C)"
    write (unit, "(7x,9A)") "import"
    write (unit, "(7x,9A)") "implicit none"
    write (unit, "(7x,9A)") "integer(c_int), intent(out) :: n"
    write (unit, "(5x,9A)") "end subroutine ", &
        char (writer%get_c_procname (id, feature))
    write (unit, "(2x,9A)") "end interface"
end subroutine test_writer_4_if

subroutine test_writer_4_wr (writer, unit, id, feature)
    class(test_writer_4_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
    write (unit, *)
    write (unit, "(9A)") "subroutine ", &
        char (writer%get_c_procname (id, feature)), &
        " (n) bind(C)"
    write (unit, "(2x,9A)") "use iso_c_binding"
    write (unit, "(2x,9A)") "use ", char (id), ", only: ", &
        char (writer%get_c_procname (feature))
    write (unit, "(2x,9A)") "implicit none"
    write (unit, "(2x,9A)") "integer(c_int), intent(out) :: n"

```

```

write (unit, "(2x,9A)") "call ", char (feature), " (n)"
write (unit, "(9A)") "end subroutine ", &
char (writer%get_c_procname (id, feature))
end subroutine test_writer_4_wr

```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless  $1 \rightarrow 2$  process, and it implements one additional function (feature), the name given as an argument.

```

<Prclib interfaces: tests>+=
subroutine write_test_module_file (basename, feature, md5sum)
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: feature
  character(32), intent(in) :: md5sum
  integer :: u
  u = free_unit ()
  open (u, file = char (basename) // ".f90", &
    status = "replace", action = "write")
  write (u, "(A)") "!(Pseudo) matrix element code file &
    &for WHIZARD self-test"
  write (u, *)
  write (u, "(A)") "module " // char (basename)
  write (u, *)
  write (u, "(2x,A)") "use kinds"
  write (u, "(2x,A)") "use omega_color, OCF => omega_color_factor"
  write (u, *)
  write (u, "(2x,A)") "implicit none"
  write (u, "(2x,A)") "private"
  write (u, *)
  call write_test_me_code_1 (u)
  write (u, *)
  write (u, "(2x,A)") "public :: " // char (feature)
  write (u, *)
  write (u, "(A)") "contains"
  write (u, *)
  call write_test_me_code_2 (u, md5sum)
  write (u, *)
  write (u, "(2x,A)") "subroutine " // char (feature) // " (n)"
  write (u, "(2x,A)") "  integer, intent(out) :: n"
  write (u, "(2x,A)") "  n = 42"
  write (u, "(2x,A)") "end subroutine " // char (feature)
  write (u, *)
  write (u, "(A)") "end module " // char (basename)
  close (u)
end subroutine write_test_module_file

```

The following two subroutines provide building blocks for a matrix-element source code file, useful only for testing the workflow. The first routine writes the header part, the other routine the implementation of the procedures listed in the header.

```

<Prclib interfaces: tests>+=
subroutine write_test_me_code_1 (u)

```

```

integer, intent(in) :: u
write (u, "(2x,A)") "public :: md5sum"
write (u, "(2x,A)") "public :: openmp_supported"
write (u, *)
write (u, "(2x,A)") "public :: n_in"
write (u, "(2x,A)") "public :: n_out"
write (u, "(2x,A)") "public :: n_flv"
write (u, "(2x,A)") "public :: n_hel"
write (u, "(2x,A)") "public :: n_cin"
write (u, "(2x,A)") "public :: n_col"
write (u, "(2x,A)") "public :: n_cf"
write (u, *)
write (u, "(2x,A)") "public :: flv_state"
write (u, "(2x,A)") "public :: hel_state"
write (u, "(2x,A)") "public :: col_state"
write (u, "(2x,A)") "public :: color_factors"
end subroutine write_test_me_code_1

subroutine write_test_me_code_2 (u, md5sum)
integer, intent(in) :: u
character(32), intent(in) :: md5sum
write (u, "(2x,A)") "pure function md5sum ()"
write (u, "(2x,A)") "  character(len=32) :: md5sum"
write (u, "(2x,A)") "  md5sum = ' ' // md5sum // '"
write (u, "(2x,A)") "end function md5sum"
write (u, *)
write (u, "(2x,A)") "pure function openmp_supported () result (status)"
write (u, "(2x,A)") "  logical :: status"
write (u, "(2x,A)") "  status = .false."
write (u, "(2x,A)") "end function openmp_supported"
write (u, *)
write (u, "(2x,A)") "pure function n_in () result (n)"
write (u, "(2x,A)") "  integer :: n"
write (u, "(2x,A)") "  n = 1"
write (u, "(2x,A)") "end function n_in"
write (u, *)
write (u, "(2x,A)") "pure function n_out () result (n)"
write (u, "(2x,A)") "  integer :: n"
write (u, "(2x,A)") "  n = 2"
write (u, "(2x,A)") "end function n_out"
write (u, *)
write (u, "(2x,A)") "pure function n_flv () result (n)"
write (u, "(2x,A)") "  integer :: n"
write (u, "(2x,A)") "  n = 1"
write (u, "(2x,A)") "end function n_flv"
write (u, *)
write (u, "(2x,A)") "pure function n_hel () result (n)"
write (u, "(2x,A)") "  integer :: n"
write (u, "(2x,A)") "  n = 1"
write (u, "(2x,A)") "end function n_hel"
write (u, *)
write (u, "(2x,A)") "pure function n_cin () result (n)"
write (u, "(2x,A)") "  integer :: n"
write (u, "(2x,A)") "  n = 2"

```

```

write (u, "(2x,A)") "end function n_cin"
write (u, *)
write (u, "(2x,A)") "pure function n_col () result (n)"
write (u, "(2x,A)") " integer :: n"
write (u, "(2x,A)") " n = 1"
write (u, "(2x,A)") "end function n_col"
write (u, *)
write (u, "(2x,A)") "pure function n_cf () result (n)"
write (u, "(2x,A)") " integer :: n"
write (u, "(2x,A)") " n = 1"
write (u, "(2x,A)") "end function n_cf"
write (u, *)
write (u, "(2x,A)") "pure subroutine flv_state (a)"
write (u, "(2x,A)") " integer, dimension(:,:), intent(out) :: a"
write (u, "(2x,A)") " a = reshape ([1,2,3], [3,1])"
write (u, "(2x,A)") "end subroutine flv_state"
write (u, *)
write (u, "(2x,A)") "pure subroutine hel_state (a)"
write (u, "(2x,A)") " integer, dimension(:,:), intent(out) :: a"
write (u, "(2x,A)") " a = reshape ([0,0,0], [3,1])"
write (u, "(2x,A)") "end subroutine hel_state"
write (u, *)
write (u, "(2x,A)") "pure subroutine col_state (a, g)"
write (u, "(2x,A)") " integer, dimension(:,:,:), intent(out) :: a"
write (u, "(2x,A)") " logical, dimension(:,:), intent(out) :: g"
write (u, "(2x,A)") " a = reshape ([0,0, 0,0, 0,0], [2,3,1])"
write (u, "(2x,A)") " g = reshape ([.false., .false., .false.], [3,1])"
write (u, "(2x,A)") "end subroutine col_state"
write (u, *)
write (u, "(2x,A)") "pure subroutine color_factors (cf)"
write (u, "(2x,A)") " type(OCF), dimension(:), intent(out) :: cf"
write (u, "(2x,A)") " cf = [ OCF(1,1,+1._default) ]"
write (u, "(2x,A)") "end subroutine color_factors"
end subroutine write_test_me_code_2

```

## Compile test with Fortran bind(C) library

Test 5: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran library of independent procedures. These procedures are bind(C).

```

<Prclib interfaces: execute tests>+≡
call test (prclib_interfaces_5, "prclib_interfaces_5", &
"compile and link (Fortran library)", &
u, results)

```

```

<Prclib interfaces: tests>+≡
subroutine prclib_interfaces_5 (u)
integer, intent(in) :: u
type(prclib_driver_t) :: driver
class(prc_writer_t), pointer :: test_writer_5
type(os_data_t) :: os_data
integer :: u_file

```

```

integer, dimension(:,:), allocatable :: flv_state
integer, dimension(:,:), allocatable :: hel_state
integer, dimension(:,:), allocatable :: col_state
logical, dimension(:,:), allocatable :: ghost_flag
integer, dimension(:,:), allocatable :: cf_index
complex(default), dimension(:), allocatable :: color_factors
character(32), parameter :: md5sum = "prclib_interfaces_5_md5sum"

type(c_funptr) :: proc1_ptr
interface
  subroutine proc1_t (n) bind(C)
    import
    integer(c_int), intent(out) :: n
  end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_5"
write (u, "(A)")  "* Purpose: compile, link, and load process library"
write (u, "(A)")  "*           with (fake) matrix-element code &
                  &as a Fortran bind(C) library"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)
allocate (test_writer_5_t :: test_writer_5)

call driver%init (var_str ("prclib5"), 1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test5"), var_str ("Test_model"), &
  [var_str ("proc1")], test_writer_5)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib5.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib5.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

```



```

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
  write (u, "(A)")  "* Call library functions:"
  write (u, *)
  write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
  write (u, "(1x,A,A)")  "process_id    = ", &
    char (driver%get_process_id (1))
  write (u, "(1x,A,A)")  "model_name    = ", &
    char (driver%get_model_name (1))
  write (u, "(1x,A,A)")  "md5sum        = ", &
    char (driver%get_md5sum (1))
  write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
  write (u, "(1x,A,I0)")  "n_in       = ", driver%get_n_in (1)
  write (u, "(1x,A,I0)")  "n_out      = ", driver%get_n_out (1)
  write (u, "(1x,A,I0)")  "n_flv      = ", driver%get_n_flv (1)
  write (u, "(1x,A,I0)")  "n_hel      = ", driver%get_n_hel (1)
  write (u, "(1x,A,I0)")  "n_col      = ", driver%get_n_col (1)
  write (u, "(1x,A,I0)")  "n_cin      = ", driver%get_n_cin (1)
  write (u, "(1x,A,I0)")  "n_cf       = ", driver%get_n_cf (1)

  call driver%set_flv_state (1, flv_state)
  write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

  call driver%set_hel_state (1, hel_state)
  write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

  call driver%set_col_state (1, col_state, ghost_flag)
  write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
  write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

  call driver%set_color_factors (1, color_factors, cf_index)
  write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
  write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

  call driver%get_fptr (1, 1, proc1_ptr)
  call c_f_procpointer (proc1_ptr, proc1)
  if (associated (proc1)) then
    write (u, *)
    call proc1 (n)
    write (u, "(1x,A,I0)")  "proc1(1) = ", n
  end if

```

```

end if

deallocate (test_writer_5)

write (u, "(A)")
write (u, "(A)")  /* Test output end: prclib_interfaces_5"
end subroutine prclib_interfaces_5

```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

```

<Prclib interfaces: test types>+≡
type, extends (prc_writer_c_lib_t) :: test_writer_5_t
contains
  procedure, nopass :: type_name => test_writer_5_type_name
  procedure :: write_makefile_code => test_writer_5_mk
  procedure :: write_source_code => test_writer_5_src
  procedure :: write_interface => test_writer_5_if
end type test_writer_5_t

```

The

```

<Prclib interfaces: tests>+≡
function test_writer_5_type_name () result (string)
  type(string_t) :: string
  string = "test_5"
end function test_writer_5_type_name

subroutine test_writer_5_mk (writer, unit, id, os_data)
  class(test_writer_5_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
  write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
  write (unit, "(5A)")  TAB, "$(LTF_COMPILE) $<"
end subroutine test_writer_5_mk

subroutine test_writer_5_src (writer, id)
  class(test_writer_5_t), intent(in) :: writer
  type(string_t), intent(in) :: id
  call write_test_f_lib_file (id, var_str ("proc1"))
end subroutine test_writer_5_src

subroutine test_writer_5_if (writer, unit, id, feature)
  class(test_writer_5_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  select case (char (feature))
  case ("proc1")
    write (unit, "(2x,9A)")  "interface"
    write (unit, "(5x,9A)")  "subroutine ", &
      char (writer%get_c_procname (id, feature)), &
      " (n) bind(C)"

```

```

write (unit, "(7x,9A)") "import"
write (unit, "(7x,9A)") "implicit none"
write (unit, "(7x,9A)") "integer(c_int), intent(out) :: n"
write (unit, "(5x,9A)") "end subroutine ", &
    char (writer%get_c_procname (id, feature))
write (unit, "(2x,9A)") "end interface"
case default
    call writer%write_standard_interface (unit, id, feature)
end select
end subroutine test_writer_5_if

```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless  $1 \rightarrow 2$  process, and it implements one additional function (feature), the name given as an argument.

*(Prclib interfaces: tests)+≡*

```

subroutine write_test_f_lib_file (basename, feature)
    type(string_t), intent(in) :: basename
    type(string_t), intent(in) :: feature
    integer :: u
    u = free_unit ()
    open (u, file = char (basename) // ".f90", &
        status = "replace", action = "write")
    write (u, "(A)") "!! (Pseudo) matrix element code file &
        &for WHIZARD self-test"
    call write_test_me_code_3 (u, char (basename))
    write (u, *)
    write (u, "(A)") "subroutine " // char (basename) // "_" &
        // char (feature) // " (n) bind(C)"
    write (u, "(A)") " use iso_c_binding"
    write (u, "(A)") " implicit none"
    write (u, "(A)") " integer(c_int), intent(out) :: n"
    write (u, "(A)") " n = 42"
    write (u, "(A)") "end subroutine " // char (basename) // "_" &
        // char (feature)
    close (u)
end subroutine write_test_f_lib_file

```

The following matrix-element source code is identical to the previous one, but modified such as to provide independent procedures without a module envelope.

*(Prclib interfaces: tests)+≡*

```

subroutine write_test_me_code_3 (u, id)
    integer, intent(in) :: u
    character(*), intent(in) :: id
    write (u, "(A)") "function " // id // "_get_md5sum () &
        &result (cptr) bind(C)"
    write (u, "(A)") " use iso_c_binding"
    write (u, "(A)") " implicit none"
    write (u, "(A)") " type(c_ptr) :: cptr"
    write (u, "(A)") " character(c_char), dimension(32), &
        &target, save :: md5sum"
    write (u, "(A)") " md5sum = copy (c_char_&
        &'1234567890abcdef1234567890abcdef')"
```

```

write (u, "(A)") " cptr = c_loc (md5sum)"
write (u, "(A)") "contains"
write (u, "(A)") " function copy (md5sum)"
write (u, "(A)") " character(c_char), dimension(32) :: copy"
write (u, "(A)") " character(c_char), dimension(32), intent(in) :: &
&md5sum"
write (u, "(A)") " copy = md5sum"
write (u, "(A)") " end function copy"
write (u, "(A)") "end function " // id // "_get_md5sum"
write (u, *)
write (u, "(A)") "function " // id // "_openmp_supported () &
&result (status) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " logical(c_bool) :: status"
write (u, "(A)") " status = .false."
write (u, "(A)") "end function " // id // "_openmp_supported"
write (u, *)
write (u, "(A)") "function " // id // "_n_in () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 1"
write (u, "(A)") "end function " // id // "_n_in"
write (u, *)
write (u, "(A)") "function " // id // "_n_out () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 2"
write (u, "(A)") "end function " // id // "_n_out"
write (u, *)
write (u, "(A)") "function " // id // "_n_flv () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 1"
write (u, "(A)") "end function " // id // "_n_flv"
write (u, *)
write (u, "(A)") "function " // id // "_n_hel () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 1"
write (u, "(A)") "end function " // id // "_n_hel"
write (u, *)
write (u, "(A)") "function " // id // "_n_cin () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 2"
write (u, "(A)") "end function " // id // "_n_cin"
write (u, *)
write (u, "(A)") "function " // id // "_n_col () result (n) bind(C)"

```

```

write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 1"
write (u, "(A)") "end function " // id // "_n_col"
write (u, *)
write (u, "(A)") "function " // id // "_n_cf () result (n) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int) :: n"
write (u, "(A)") " n = 1"
write (u, "(A)") "end function " // id // "_n_cf"
write (u, *)
write (u, "(A)") "subroutine " // id // "_flv_state (flv_state) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int), dimension(*), intent(out) :: flv_state"
write (u, "(A)") " flv_state(1:3) = [1,2,3]"
write (u, "(A)") "end subroutine " // id // "_flv_state"
write (u, *)
write (u, "(A)") "subroutine " // id // "_hel_state (hel_state) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int), dimension(*), intent(out) :: hel_state"
write (u, "(A)") " hel_state(1:3) = [0,0,0]"
write (u, "(A)") "end subroutine " // id // "_hel_state"
write (u, *)
write (u, "(A)") "subroutine " // id // "_col_state &
&(col_state, ghost_flag) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int), dimension(*), intent(out) &
&:: col_state"
write (u, "(A)") " logical(c_bool), dimension(*), intent(out) &
&:: ghost_flag"
write (u, "(A)") " col_state(1:6) = [0,0, 0,0, 0,0]"
write (u, "(A)") " ghost_flag(1:3) = [.false., .false., .false.]"
write (u, "(A)") "end subroutine " // id // "_col_state"
write (u, *)
write (u, "(A)") "subroutine " // id // "_color_factors &
&(cf_index1, cf_index2, color_factors) bind(C)"
write (u, "(A)") " use iso_c_binding"
write (u, "(A)") " use kinds"
write (u, "(A)") " implicit none"
write (u, "(A)") " integer(c_int), dimension(*), intent(out) :: cf_index1"
write (u, "(A)") " integer(c_int), dimension(*), intent(out) :: cf_index2"
write (u, "(A)") " complex(c_default_complex), dimension(*), &
&intent(out) :: color_factors"
write (u, "(A)") " cf_index1(1:1) = [1]"
write (u, "(A)") " cf_index2(1:1) = [1]"
write (u, "(A)") " color_factors(1:1) = [1]"
write (u, "(A)") "end subroutine " // id // "_color_factors"
end subroutine write_test_me_code_3

```

## Compile test with genuine C library

Test 6: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a C library of independent procedures. These procedures should match the Fortran bind(C) interface.

```
(Prclib interfaces: execute tests)+≡
    call test (prclib_interfaces_6, "prclib_interfaces_6", &
               "compile and link (C library)", &
               u, results)

(Prclib interfaces: tests)+≡
subroutine prclib_interfaces_6 (u)
    integer, intent(in) :: u
    type(prclib_driver_t) :: driver
    class(prc_writer_t), pointer :: test_writer_6
    type(os_data_t) :: os_data
    integer :: u_file

    integer, dimension(:,:), allocatable :: flv_state
    integer, dimension(:,:), allocatable :: hel_state
    integer, dimension(:,:), allocatable :: col_state
    logical, dimension(:,:), allocatable :: ghost_flag
    integer, dimension(:,:), allocatable :: cf_index
    complex(default), dimension(:), allocatable :: color_factors
    character(32), parameter :: md5sum = "prclib_interfaces_6_md5sum"

    type(c_funptr) :: proc1_ptr
    interface
        subroutine proc1_t (n) bind(C)
            import
            integer(c_int), intent(out) :: n
        end subroutine proc1_t
    end interface
    procedure(proc1_t), pointer :: proc1
    integer(c_int) :: n

    write (u, "(A)")  "* Test output: prclib_interfaces_6"
    write (u, "(A)")  "* Purpose: compile, link, and load process library"
    write (u, "(A)")  "*           with (fake) matrix-element code &
                        &as a C library"
    write (u, *)
    write (u, "(A)")  "* Create a prclib driver object (1 process)"
    write (u, "(A)")

    call os_data_init (os_data)
    allocate (test_writer_6_t :: test_writer_6)

    call driver%init (var_str ("prclib6"), 1)
    call driver%set_md5sum (md5sum)

    call driver%set_record (1, var_str ("test6"), var_str ("Test_model"), &
                           [var_str ("proc1")], test_writer_6)

    call driver%write (u)
```

```

write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib6.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib6.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
  write (u, "(A)")  "* Call library functions:"
  write (u, *)
  write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
  write (u, "(1x,A,A)")  "process_id   = ", &
    char (driver%get_process_id (1))
  write (u, "(1x,A,A)")  "model_name   = ", &
    char (driver%get_model_name (1))
  write (u, "(1x,A,A)")  "md5sum       = ", &
    char (driver%get_md5sum (1))
  write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
  write (u, "(1x,A,I0)")  "n_in        = ", driver%get_n_in (1)
  write (u, "(1x,A,I0)")  "n_out       = ", driver%get_n_out (1)
  write (u, "(1x,A,I0)")  "n_flv       = ", driver%get_n_flv (1)
  write (u, "(1x,A,I0)")  "n_hel       = ", driver%get_n_hel (1)
  write (u, "(1x,A,I0)")  "n_col       = ", driver%get_n_col (1)
  write (u, "(1x,A,I0)")  "n_cin       = ", driver%get_n_cin (1)
  write (u, "(1x,A,I0)")  "n_cf        = ", driver%get_n_cf (1)

  call driver%set_flv_state (1, flv_state)
  write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

  call driver%set_hel_state (1, hel_state)
  write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

```

```

call driver%set_col_state (1, col_state, ghost_flag)
write (u, "(1x,A,10(1x,I0))") "col_state =", col_state
write (u, "(1x,A,10(1x,L1))") "ghost_flag =", ghost_flag

call driver%set_color_factors (1, color_factors, cf_index)
write (u, "(1x,A,10(1x,F5.3))") "color_factors =", color_factors
write (u, "(1x,A,10(1x,I0))") "cf_index =", cf_index

call driver%get_fptr (1, 1, proc1_ptr)
call c_f_procpointer (proc1_ptr, proc1)
if (associated (proc1)) then
  write (u, *)
  call proc1 (n)
  write (u, "(1x,A,I0)") "proc1(1) = ", n
end if

end if

deallocate (test_writer_6)

write (u, "(A)")
write (u, "(A)")  "* Test output end: prclib_interfaces_6"
end subroutine prclib_interfaces_6

```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

The driver part is identical to the Fortran case, so we simply extend the previous `test_writer_5` type. We only have to override the Makefile writer.

```

<Prclib interfaces: test types>+=
type, extends (test_writer_5_t) :: test_writer_6_t
contains
  procedure, nopass :: type_name => test_writer_6_type_name
  procedure :: write_makefile_code => test_writer_6_mk
  procedure :: write_source_code => test_writer_6_src
end type test_writer_6_t

<Prclib interfaces: tests>+=
function test_writer_6_type_name () result (string)
  type(string_t) :: string
  string = "test_6"
end function test_writer_6_type_name

subroutine test_writer_6_mk (writer, unit, id, os_data)
  class(test_writer_6_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  write (unit, "(5A)") "SOURCES += ", char (id), ".c"
  write (unit, "(5A)") "OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)") char (id), ".lo: ", char (id), ".c"
  write (unit, "(5A)") TAB, "$(LTCCOMPILER) $<"
end subroutine test_writer_6_mk

```



```

subroutine test_writer_6_src (writer, id)
  class(test_writer_6_t), intent(in) :: writer
  type(string_t), intent(in) :: id
!   call write_test_c_header_file (id, var_str ("proc1"))
   call write_test_c_lib_file (id, var_str ("proc1"))
end subroutine test_writer_6_src

```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless  $1 \rightarrow 2$  process, and it implements one additional function (feature), the name given as an argument.

```

(Prclib interfaces: tests)+=
subroutine write_test_c_lib_file (basename, feature)
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: feature
  integer :: u
  u = free_unit ()
  open (u, file = char (basename) // ".c", &
        status = "replace", action = "write")
  write (u, "(A)")  "/* (Pseudo) matrix element code file &
    &for WHIZARD self-test */"
  write (u, "(A)")  "#include <stdbool.h>"
  write (u, *)
  call write_test_me_code_4 (u, char (basename))
  write (u, *)
  write (u, "(A)")  "void " // char (basename) // "_" &
    // char (feature) // "(int* n) {"
  write (u, "(A)")  "  *n = 42;"
  write (u, "(A)")  "}"
  close (u)
end subroutine write_test_c_lib_file

```

The following matrix-element source code is equivalent to the code in the previous example, but coded in C.

```

(Prclib interfaces: tests)+=
subroutine write_test_me_code_4 (u, id)
  integer, intent(in) :: u
  character(*), intent(in) :: id
  write (u, "(A)")  "char* " // id // "_get_md5sum() {"
  write (u, "(A)")  "  return ""1234567890abcdef1234567890abcdef"";"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "bool " // id // "_openmp_supported() {"
  write (u, "(A)")  "  return false;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_in() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_out() {"
  write (u, "(A)")  "  return 2;"

```

```

write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_flv() {"
write (u, "(A)")  "    return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_hel() {"
write (u, "(A)")  "    return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_cin() {"
write (u, "(A)")  "    return 2;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_col() {"
write (u, "(A)")  "    return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_cf() {"
write (u, "(A)")  "    return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_flv_state( int (*a)[] ) {"
write (u, "(A)")  "    static int flv_state[1][3] = { { 1, 2, 3 } };"
write (u, "(A)")  "    int j;"
write (u, "(A)")  "    for (j = 0; j < 3; j++) { (*a)[j] &
        &= flv_state[0][j]; }"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_hel_state( int (*a)[] ) {"
write (u, "(A)")  "    static int hel_state[1][3] = { { 0, 0, 0 } };"
write (u, "(A)")  "    int j;"
write (u, "(A)")  "    for (j = 0; j < 3; j++) { (*a)[j] &
        &= hel_state[0][j]; }"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_col_state&
        &( int (*a)[], bool (*g)[] ) {"
write (u, "(A)")  "    static int col_state[1][3][2] = &
        &{ { {0, 0}, {0, 0}, {0, 0} } };"
write (u, "(A)")  "    static bool ghost_flag[1][3] = &
        &{ { false, false, false } };"
write (u, "(A)")  "    int j,k;"
write (u, "(A)")  "    for (j = 0; j < 3; j++) {"
write (u, "(A)")  "        for (k = 0; k < 2; k++) {"
write (u, "(A)")  "            (*a)[j*2+k] = col_state[0][j][k];"
write (u, "(A)")  "        }"
write (u, "(A)")  "        (*g)[j] = ghost_flag[0][j];"
write (u, "(A)")  "    }"
write (u, "(A)")  "}"
write (u, *)
if (c_default_complex == c_long_double_complex) then
    write (u, "(A)")  "void " // id // "_color_factors&
        &( int (*cf_index1)[], int (*cf_index2)[], &

```

```

        &long double _Complex (*color_factors)[] ) {"
else
    write (u, "(A)") "void " // id // "_color_factors&
        &( int (*cf_index1)[], int (*cf_index2)[], &
        &double _Complex (*color_factors)[] ) {"
end if
write (u, "(A)") " (*color_factors)[0] = 1;"
write (u, "(A)") " (*cf_index1)[0] = 1;"
write (u, "(A)") " (*cf_index2)[0] = 1;"
write (u, "(A)") "}"
end subroutine write_test_me_code_4

```

## Test cleanup targets

Test 7: Repeat test 4 (create, compile, link Fortran module and driver) and properly clean up all generated files.

```

<Prclib interfaces: execute tests>+≡
    call test (prclib_interfaces_7, "prclib_interfaces_7", &
        "cleanup", &
        u, results)

<Prclib interfaces: tests>+≡
subroutine prclib_interfaces_7 (u)
    integer, intent(in) :: u
    type(prclib_driver_t) :: driver
    class(prc_writer_t), pointer :: test_writer_4
    type(os_data_t) :: os_data
    integer :: u_file
    character(32), parameter :: md5sum = "1234567890abcdef1234567890abcdef"

    write (u, "(A)") "* Test output: prclib_interfaces_7"
    write (u, "(A)") "* Purpose: compile and link process library"
    write (u, "(A)") "*           with (fake) matrix-element code &
        &as a Fortran module"
    write (u, "(A)") "*           then clean up generated files"
    write (u, *)
    write (u, "(A)") "* Create a prclib driver object (1 process)"

    allocate (test_writer_4_t :: test_writer_4)

    call os_data_init (os_data)
    call driver%init (var_str ("prclib7"), 1)
    call driver%set_md5sum (md5sum)
    call driver%set_record (1, var_str ("test7"), var_str ("Test_model"), &
        [var_str ("proc1")], test_writer_4)

    write (u, "(A)") "* Write makefile"
    u_file = free_unit ()
    open (u_file, file="prclib7.makefile", status="replace", action="write")
    call driver%generate_makefile (u_file, os_data)
    close (u_file)

    write (u, "(A)") "* Write driver source code"

```

```

u_file = free_unit ()
open (u_file, file="prclib7.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)


write (u, "(A)")  "* File check"
write (u, *)
call check_file (u, "test7.f90")
call check_file (u, "test7.mod")
call check_file (u, "test7.lo")
call check_file (u, "prclib7.makefile")
call check_file (u, "prclib7.f90")
call check_file (u, "prclib7.lo")
call check_file (u, "prclib7.la")

write (u, *)
write (u, "(A)")  "* Delete library"
write (u, *)
call driver%clean_library (os_data)
call check_file (u, "prclib7.la")

write (u, *)
write (u, "(A)")  "* Delete object code"
write (u, *)
call driver%clean_objects (os_data)
call check_file (u, "test7.lo")
call check_file (u, "test7.mod")
call check_file (u, "prclib7.lo")

write (u, *)
write (u, "(A)")  "* Delete source code"
write (u, *)
call driver%clean_source (os_data)
call check_file (u, "test7.f90")

write (u, *)
write (u, "(A)")  "* Delete driver source code"
write (u, *)
call driver%clean_driver (os_data)
call check_file (u, "prclib7.f90")

write (u, *)
write (u, "(A)")  "* Delete makefile"
write (u, *)

```

```

call driver%clean_makefile (os_data)
call check_file (u, "prclib7.makefile")

deallocate (test_writer_4)

write (u, *)
write (u, "(A)")  "* Test output end: prclib_interfaces_7"
end subroutine prclib_interfaces_7

```

Auxiliary routine: check and report existence of a file

```

<Prclib interfaces: tests>+=
subroutine check_file (u, file)
  integer, intent(in) :: u
  character(*), intent(in) :: file
  logical :: exist
  inquire (file=file, exist=exist)
  write (u, "(2x,A,A,L1)")  file, " = ", exist
end subroutine check_file

```

## 14.2 Abstract process core configuration

In this module, we define abstract data types that handle the method-specific part of defining a process (including all of its options) and accessing an external matrix element.

There are no unit tests, these are deferred to the `process_libraries` module below.

```

<prc_core_def.f90>=
<File header>

module prc_core_def

  <Use strings>
  <Use file utils>
  use diagnostics !NODEP!

  use process_constants
  use prclib_interfaces

  <Standard module head>

  <Prc core def: public>

  <Prc core def: types>

  <Prc core def: interfaces>

  contains

  <Prc core def: procedures>

end module prc_core_def

```

### 14.2.1 Process core definition type

For storing configuration data that depend on the specific process variant, we introduce a polymorphic type. At this point, we just declare an abstract base type. This allows us to defer the implementation to later modules.

There should be no components that need explicit finalization, otherwise we would have to call a finalizer from the `process_component_def_t` wrapper.

```
<Prc core def: public>≡
    public :: prc_core_def_t

<Prc core def: types>≡
    type, abstract :: prc_core_def_t
        class(prc_writer_t), allocatable :: writer
        contains
        <Prc core def: process core def: TBP>
    end type prc_core_def_t
```

Interfaces for the deferred methods.

This returns a string. No passed argument; the string is constant and depends just on the type.

```
<Prc core def: process core def: TBP>≡
    procedure (prc_core_def_get_string), nopass, deferred :: type_string

<Prc core def: interfaces>≡
    abstract interface
        function prc_core_def_get_string () result (string)
            import
            type(string_t) :: string
        end function prc_core_def_get_string
    end interface
```

The `write` method should display the content completely.

```
<Prc core def: process core def: TBP>+≡
    procedure (prc_core_def_write), deferred :: write

<Prc core def: interfaces>+≡
    abstract interface
        subroutine prc_core_def_write (object, unit)
            import
            class(prc_core_def_t), intent(in) :: object
            integer, intent(in) :: unit
        end subroutine prc_core_def_write
    end interface
```

The `read` method should fill the content completely.

```
<Prc core def: process core def: TBP>+≡
    procedure (prc_core_def_read), deferred :: read

<Prc core def: interfaces>+≡
    abstract interface
        subroutine prc_core_def_read (object, unit)
            import
            class(prc_core_def_t), intent(out) :: object
            integer, intent(in) :: unit
```

```

        end subroutine prc_core_def_read
    end interface

```

This communicates a MD5 checksum to the writer inside the `core_def` object, if there is any. Usually, this checksum is not yet known at the time when the writer is initialized.

```

<Prc core def: process core def: TBP>+≡
    procedure :: set_md5sum => prc_core_def_set_md5sum

<Prc core def: procedures>≡
    subroutine prc_core_def_set_md5sum (core_def, md5sum)
        class(prc_core_def_t), intent(inout) :: core_def
        character(32) :: md5sum
        if (allocated (core_def%writer)) core_def%writer%md5sum = md5sum
    end subroutine prc_core_def_set_md5sum

```

Allocate an appropriate driver object which corresponds to the chosen process core definition.

For internal matrix element (i.e., those which do not need external code), the driver should have access to all matrix element information from the beginning. In short, it is the matrix-element code.

For external matrix elements, the driver will get access to the external matrix element code.

```

<Prc core def: process core def: TBP>+≡
    procedure(prc_core_def_allocate_driver), deferred :: allocate_driver

<Prc core def: interfaces>+≡
    interface
        subroutine prc_core_def_allocate_driver (object, driver, basename)
            import
            class(prc_core_def_t), intent(in) :: object
            class(prc_core_driver_t), intent(out), allocatable :: driver
            type(string_t), intent(in) :: basename
        end subroutine prc_core_def_allocate_driver
    end interface

```

This flag tells whether the particular variant needs external code. We implement a default function which returns false. The flag depends only on the type, therefore we implement it as `nopass`.

```

<Prc core def: process core def: TBP>+≡
    procedure, nopass :: needs_code => prc_core_def_needs_code

<Prc core def: procedures>+≡
    function prc_core_def_needs_code () result (flag)
        logical :: flag
        flag = .false.
    end function prc_core_def_needs_code

```

This subroutine allocates an array which holds the name of all features that this process core implements. This feature applies to matrix element code that is not coded as a Fortran module but communicates via independent library functions, which follow the C calling conventions. The addresses of those functions are

returned as C function pointers, which can be converted into Fortran procedure pointers. The conversion is done in code specific for the process variant; here we just retrieve the C function pointer.

The array returned here serves the purpose of writing specific driver code. The driver interfaces only those C functions which are supported for the given process core.

If the process core does not require external code, this array is meaningless.

```

<Prc core def: process core def: TBP>+≡
  procedure(prc_core_def_get_features), nopass, deferred &
    :: get_features

<Prc core def: interfaces>+≡
  abstract interface
    subroutine prc_core_def_get_features (features)
      import
      type(string_t), dimension(:), allocatable, intent(out) :: features
    end subroutine prc_core_def_get_features
  end interface

```

Assign pointers to the process-specific procedures to the driver, if the process is external.

```

<Prc core def: process core def: TBP>+≡
  procedure(prc_core_def_connect), deferred :: connect

<Prc core def: interfaces>+≡
  abstract interface
    subroutine prc_core_def_connect (def, lib_driver, i, proc_driver)
      import
      class(prc_core_def_t), intent(in) :: def
      type(prclib_driver_t), intent(in) :: lib_driver
      integer, intent(in) :: i
      class(prc_core_driver_t), intent(inout) :: proc_driver
    end subroutine prc_core_def_connect
  end interface

```

## 14.2.2 Process core template

We must be able to automatically allocate a process core definition object with the appropriate type, given only the type name.

To this end, we introduce a `prc_template_t` type which is simply a wrapper for an empty `prc_core_def_t` object. Choosing one of the templates from an array, we can allocate the target object.

```

<Prc core def: public>+≡
  public :: prc_template_t

<Prc core def: types>+≡
  type :: prc_template_t
    class(prc_core_def_t), allocatable :: core_def
  end type prc_template_t

```



The allocation routine. We use the `source` option of the `allocate` statement. The `mold` option would probably more appropriate, but is a F2008 feature.

```

<Prc core def: public>+≡
    public :: allocate_core_def

<Prc core def: procedures>+≡
    subroutine allocate_core_def (template, name, core_def)
        type(prc_template_t), dimension(:), intent(in) :: template
        type(string_t), intent(in) :: name
        class(prc_core_def_t), allocatable :: core_def
        integer :: i
        do i = 1, size (template)
            if (template(i)%core_def%type_string () == name) then
                allocate (core_def, source = template(i)%core_def)
                return
            end if
        end do
    end subroutine allocate_core_def

```

### 14.2.3 Process driver

For each process component, we implement a driver object which holds the calls to the matrix element and various auxiliary routines as procedure pointers. Any actual calculation will use this object to communicate with the process.

Depending on the type of process (as described by a corresponding `prc_core_def` object), the procedure pointers may refer to external or internal code, and there may be additional procedures for certain types. The base type defined here is abstract.

```

<Prc core def: public>+≡
    public :: prc_core_driver_t

<Prc core def: types>+≡
    type, abstract :: prc_core_driver_t
        contains
        <Prc core def: process driver: TBP>
    end type prc_core_driver_t

```

This returns the process type. No reference to contents.

```

<Prc core def: process driver: TBP>≡
    procedure(prc_core_driver_type_name), nopass, deferred :: type_name

<Prc core def: interfaces>+≡
    abstract interface
        function prc_core_driver_type_name () result (type)
            import
            type(string_t) :: type
        end function prc_core_driver_type_name
    end interface

```

### 14.2.4 Process driver for intrinsic process

This is an abstract extension for the driver type. It has one additional method, namely a subroutine that fills the record of constant process data. For an external process, this task is performed by the external library driver instead.

```

<Prc core def: public>+≡
    public :: process_driver_internal_t

<Prc core def: types>+≡
    type, extends (prc_core_driver_t), abstract :: process_driver_internal_t
    contains
    <Prc core def: process driver internal: TBP>
    end type process_driver_internal_t

<Prc core def: process driver internal: TBP>≡
    procedure(process_driver_fill_constants), deferred :: fill_constants

<Prc core def: interfaces>+≡
    abstract interface
        subroutine process_driver_fill_constants (driver, data)
            import
            class(process_driver_internal_t), intent(in) :: driver
            type(process_constants_t), intent(out) :: data
        end subroutine process_driver_fill_constants
    end interface

```

## 14.3 Particle Specifiers

In this module we introduce a type for specifying a particle or particle alternative. In addition to the particle specifiers (strings separated by colons), the type contains an optional flag **polarized** and a string **decay**. If the **polarized** flag is set, particle polarization information should be kept when generating events for this process. If the **decay** string is set, it is the ID of a decay process which should be applied to this particle when generating events.

In input/output form, the **polarized** flag is indicated by an asterisk (\*) in brackets, and the **decay** is indicated by its ID in brackets.

The **read** and **write** procedures in this module are not type-bound but generic procedures which handle scalar and array arguments.

```

<particle_specifiers.f90>≡
    <File header>

    module particle_specifiers

    <Use strings>
    <Use file utils>
        use diagnostics !NODEP!
        use unit_tests

    <Standard module head>

    <Particle specifiers: public>

```

```

    <Particle specifiers: types>

    <Particle specifiers: interfaces>

contains

    <Particle specifiers: procedures>

    <Particle specifiers: tests>

end module particle_specifiers

```

### 14.3.1 The type

The particle is unstable if the `decay` array is allocated. The `polarized` flag and decays may not be set simultaneously.

```

<Particle specifiers: public>≡
    public :: prt_spec_t

<Particle specifiers: types>≡
    type :: prt_spec_t
        private
        type(string_t) :: name
        logical :: polarized = .false.
        type(string_t), dimension(:), allocatable :: decay
    contains
        <Particle specifiers: prt spec: TBP>
    end type prt_spec_t

```

Output.

```

<Particle specifiers: public>+≡
    public :: prt_spec_write

<Particle specifiers: interfaces>≡
    interface prt_spec_write
        module procedure prt_spec_write1
        module procedure prt_spec_write2
    end interface prt_spec_write

<Particle specifiers: procedures>≡
    subroutine prt_spec_write1 (object, unit, advance)
        type(prt_spec_t), intent(in) :: object
        integer, intent(in), optional :: unit
        character(len=*), intent(in), optional :: advance
        character(3) :: adv
        integer :: u
        u = output_unit (unit)
        adv = "yes"; if (present (advance)) adv = advance
        write (u, "(A)", advance = trim (adv)) char (object%to_string ())
    end subroutine prt_spec_write1

```

Write an array as a list of particle specifiers.

```

<Particle specifiers: procedures>+≡
subroutine prt_spec_write2 (prt_spec, unit, advance)
  type(prt_spec_t), dimension(:), intent(in) :: prt_spec
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: advance
  character(3) :: adv
  integer :: u, i
  u = output_unit (unit)
  adv = "yes"; if (present (advance)) adv = advance
  do i = 1, size (prt_spec)
    if (i > 1) write (u, "(A)", advance="no") " , "
    call prt_spec_write (prt_spec(i), u, advance="no")
  end do
  write (u, "(A)", advance = trim (adv))
end subroutine prt_spec_write2

```

Read. Input may be string or array of strings.

```

<Particle specifiers: public>+≡
public :: prt_spec_read

<Particle specifiers: interfaces>+≡
interface prt_spec_read
  module procedure prt_spec_read1
  module procedure prt_spec_read2
end interface prt_spec_read

```

Read a single particle specifier

```

<Particle specifiers: procedures>+≡
pure subroutine prt_spec_read1 (prt_spec, string)
  type(prt_spec_t), intent(out) :: prt_spec
  type(string_t), intent(in) :: string
  type(string_t) :: arg, buffer
  integer :: b1, b2, c, n, i
  b1 = scan (string, "(")
  b2 = scan (string, ")")
  if (b1 == 0) then
    prt_spec%name = trim (adjustl (string))
  else
    prt_spec%name = trim (adjustl (extract (string, 1, b1-1)))
    arg = trim (adjustl (extract (string, b1+1, b2-1)))
    if (arg == "*") then
      prt_spec%polarized = .true.
    else
      n = 0
      buffer = arg
      do
        n = n + 1
        c = scan (buffer, "+")
        if (c == 0) exit
        buffer = extract (buffer, c+1)
      end do
      allocate (prt_spec%decay (n))
      buffer = arg
    end if
  end if
end subroutine prt_spec_read1

```

```

do i = 1, n
  c = scan (buffer, "+")
  if (c == 0) c = len (buffer) + 1
  prt_spec%decay(i) = trim (adjustl (extract (buffer, 1, c-1)))
  buffer = extract (buffer, c+1)
end do
end if
end if
end subroutine prt_spec_read1

```

Read a particle specifier array, given as a single string. The array is allocated to the correct size.

*(Particle specifiers: procedures)*+≡

```

pure subroutine prt_spec_read2 (prt_spec, string)
  type(prt_spec_t), dimension(:), intent(out), allocatable :: prt_spec
  type(string_t), intent(in) :: string
  type(string_t) :: buffer
  integer :: c, i, n
  n = 0
  buffer = string
  do
    n = n + 1
    c = scan (buffer, ",")
    if (c == 0) exit
    buffer = extract (buffer, c+1)
  end do
  allocate (prt_spec (n))
  buffer = string
  do i = 1, size (prt_spec)
    c = scan (buffer, ",")
    if (c == 0) c = len (buffer) + 1
    call prt_spec_read (prt_spec(i), &
      trim (adjustl (extract (buffer, 1, c-1))))
    buffer = extract (buffer, c+1)
  end do
end subroutine prt_spec_read2

```

### 14.3.2 Constructor

Initialize a particle specifier.

*(Particle specifiers: public)*+≡

```

public :: new_prt_spec

```

*(Particle specifiers: interfaces)*+≡

```

interface new_prt_spec
  module procedure new_prt_spec
  module procedure new_prt_spec_polarized
  module procedure new_prt_spec_unstable
end interface new_prt_spec

```

*(Particle specifiers: procedures)*+≡

```

elemental function new_prt_spec (name) result (prt_spec)
  type(string_t), intent(in) :: name

```

```

    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
end function new_prt_spec

elemental function new_prt_spec_polarized (name, polarized) result (prt_spec)
    type(string_t), intent(in) :: name
    logical, intent(in) :: polarized
    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
    prt_spec%polarized = polarized
end function new_prt_spec_polarized

pure function new_prt_spec_unstable (name, decay) result (prt_spec)
    type(string_t), intent(in) :: name
    type(string_t), dimension(:), intent(in) :: decay
    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
    allocate (prt_spec%decay (size (decay)))
    prt_spec%decay = decay
end function new_prt_spec_unstable

```

### 14.3.3 Access Methods

Return the particle name without qualifiers

```

<Particle specifiers: prt spec: TBP>≡
    procedure :: get_name => prt_spec_get_name

<Particle specifiers: procedures>+≡
    elemental function prt_spec_get_name (prt_spec) result (name)
        class(prt_spec_t), intent(in) :: prt_spec
        type(string_t) :: name
        name = prt_spec%name
    end function prt_spec_get_name

```

Return the name with qualifiers

```

<Particle specifiers: prt spec: TBP>+≡
    procedure :: to_string => prt_spec_to_string

<Particle specifiers: procedures>+≡
    elemental function prt_spec_to_string (prt_spec) result (string)
        class(prt_spec_t), intent(in) :: prt_spec
        type(string_t) :: string
        integer :: i
        string = prt_spec%name
        if (allocated (prt_spec%decay)) then
            string = string // "("
            do i = 1, size (prt_spec%decay)
                if (i > 1) string = string // " + "
                string = string // prt_spec%decay(i)
            end do
            string = string // ")"
        else if (prt_spec%polarized) then
            string = string // "(*)"
        end if
    end function prt_spec_to_string

```

```

        end if
    end function prt_spec_to_string

```

Return the polarization flag

```

<Particle specifiers: prt spec: TBP>+≡
    procedure :: is_polarized => prt_spec_is_polarized

<Particle specifiers: procedures>+≡
    elemental function prt_spec_is_polarized (prt_spec) result (flag)
        class(prt_spec_t), intent(in) :: prt_spec
        logical :: flag
        flag = prt_spec%polarized
    end function prt_spec_is_polarized

```

The particle is unstable if there is a decay array.

```

<Particle specifiers: prt spec: TBP>+≡
    procedure :: is_unstable => prt_spec_is_unstable

<Particle specifiers: procedures>+≡
    elemental function prt_spec_is_unstable (prt_spec) result (flag)
        class(prt_spec_t), intent(in) :: prt_spec
        logical :: flag
        flag = allocated (prt_spec%decay)
    end function prt_spec_is_unstable

```

Return the number of decay channels

```

<Particle specifiers: prt spec: TBP>+≡
    procedure :: get_n_decays => prt_spec_get_n_decays

<Particle specifiers: procedures>+≡
    elemental function prt_spec_get_n_decays (prt_spec) result (n)
        class(prt_spec_t), intent(in) :: prt_spec
        integer :: n
        if (allocated (prt_spec%decay)) then
            n = size (prt_spec%decay)
        else
            n = 0
        end if
    end function prt_spec_get_n_decays

```

Return the decay channels

```

<Particle specifiers: prt spec: TBP>+≡
    procedure :: get_decays => prt_spec_get_decays

<Particle specifiers: procedures>+≡
    subroutine prt_spec_get_decays (prt_spec, decay)
        class(prt_spec_t), intent(in) :: prt_spec
        type(string_t), dimension(:), allocatable, intent(out) :: decay
        if (allocated (prt_spec%decay)) then
            allocate (decay (size (prt_spec%decay)))
            decay = prt_spec%decay
        else
            allocate (decay (0))
        end if
    end subroutine prt_spec_get_decays

```

```

        end if
    end subroutine prt_spec_get_decays

```

#### 14.3.4 Test

This is the master for calling self-test procedures.

```

<Particle specifiers: public>+≡
    public :: particle_specifiers_test

<Particle specifiers: tests>≡
    subroutine particle_specifiers_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Particle specifiers: execute tests>
    end subroutine particle_specifiers_test

```

#### Empty process list

Test 1: Write an empty process list.

```

<Particle specifiers: execute tests>≡
    call test (particle_specifiers_1, "particle_specifiers_1", &
        "Handle particle specifiers", &
        u, results)

<Particle specifiers: tests>+≡
    subroutine particle_specifiers_1 (u)
        integer, intent(in) :: u
        type(prt_spec_t), dimension(:), allocatable :: prt_spec
        type(string_t), dimension(:), allocatable :: decay
        integer :: i, j

        write (u, "(A)")  "* Test output: particle_specifiers_1"
        write (u, "(A)")  "* Purpose: Read and write a particle specifier array"
        write (u, "(A)")

        allocate (prt_spec (4))
        prt_spec = [ &
            new_prt_spec (var_str ("a")), &
            new_prt_spec (var_str ("b"), .true.), &
            new_prt_spec (var_str ("c"), [var_str ("dec1")]), &
            new_prt_spec (var_str ("d"), [var_str ("dec1"), var_str ("dec2")]) &
        ]
        do i = 1, size (prt_spec)
            write (u, "(A)")  char (prt_spec(i)%to_string ())
        end do
        write (u, "(A)")

        call prt_spec_read (prt_spec, &
            var_str (" a, b( *), c( dec1), d (dec1 + dec2 )"))
        call prt_spec_write (prt_spec, u)

        do i = 1, size (prt_spec)

```



```

write (u, "(A)")
write (u, "(A,A)") char (prt_spec(i)%get_name ()), ":"
write (u, "(A,L1)") "polarized = ", prt_spec(i)%is_polarized ()
write (u, "(A,L1)") "unstable = ", prt_spec(i)%is_unstable ()
write (u, "(A,I0)") "n_decays = ", prt_spec(i)%get_n_decays ()
call prt_spec(i)%get_decays (decay)
write (u, "(A)", advance="no") "decays    ="
do j = 1, size (decay)
    write (u, "(1x,A)", advance="no") char (decay(j))
end do
write (u, "(A)")
end do

write (u, "(A)")
write (u, "(A)")  "* Test output end: particle_specifiers_1"
end subroutine particle_specifiers_1

```

## 14.4 Process library access

Processes (the code and data that are necessary for evaluating matrix elements of a particular process or process component) are organized in process libraries. In full form, process libraries contain generated and dynamically compiled and linked code, so they are actual libraries on the OS level. Alternatively, there may be simple processes that can be generated without referring to external libraries, and external libraries that are just linked in.

This module interfaces the OS to create, build, and use process libraries.

We work with two related data structures. There is the list of process configurations that stores the user input and data derived from it. A given process configuration list is scanned for creating a process library, which consists of both data and code. The creation step involves calling external programs and incorporating external code.

For the subsequent integration and event generation steps, we read the process library. We also support partial (re)creation of the process library. To this end, we should be able to reconstruct the configuration data records from the process library.

```

<process_libraries.f90>≡
<File header>

module process_libraries

    use iso_c_binding !NODEP!
    use kinds !NODEP!
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use md5
    use unit_tests
    use os_interface
    use lexers
    use variables

```

```

    use user_code_interface
    use models
    use flavors

    use process_constants
    use prclib_interfaces
    use prc_core_def
    use particle_specifiers

    <Standard module head>

    <Process libraries: public>

    <Process libraries: parameters>

    <Process libraries: types>

    <Process libraries: interfaces>

    <Process libraries: variables>

    <Process libraries: test declarations>

contains

    <Process libraries: procedures>

    <Process libraries: tests>

end module process_libraries

```

#### 14.4.1 Auxiliary stuff

Here is a small subroutine that strips the left-hand side and the equals sign off an equation.

```

<Process libraries: procedures>≡
    subroutine strip_prefix (buffer)
        character(*), intent(inout) :: buffer
        type(string_t) :: string, prefix
        string = buffer
        call split (string, prefix, "=")
        buffer = string
    end subroutine strip_prefix

```

#### 14.4.2 Process definition objects

We collect process configuration data in a derived type, `process_def_t`. A process can be a collection of several components which are treated as a single entity for the purpose of observables and event generation. Multiple process components may initially be defined by the user. The system may add additional components, e.g., subtraction terms. The common data type is

`process_component_def_t`. Within each component, there are several universal data items, and a part which depend on the particular process variant. The latter is covered by an abstract type `prc_core_def_t` and its extensions.

### Wrapper for components

We define a wrapper type for the configuration of individual components.

The string `basename` is used for building file, module, and function names for the current process component. Initially, it will be built from the corresponding process basename by appending an alphanumeric suffix.

The logical `initial` tells whether this is a user-defined (true) or system-generated (false) configuration.

The numbers `n_in`, `n_out`, and `n_tot` denote the incoming, outgoing and total number of particles (partons) participating in the process component, respectively. These are the nominal particles, as input by the user (recombination may change the particle content, for the output events).

The string arrays `prt_in` and `prt_out` hold the particle specifications as provided by the user. For a system-generated process component, they remain deallocated.

The `method` string is used to determine the type of process matrix element and how it is obtained.

The `description` string collects the information about particle content and method in a single human-readable string.

The pointer object `core_def` is allocated according to the actual process variant, which depends on the method. The subobject holds any additional configuration data that is relevant for the process component.

We assume that no finalizer is needed.

```

(Process libraries: public)≡
    public :: process_component_def_t

(Process libraries: types)≡
    type :: process_component_def_t
    private
        type(string_t) :: basename
        logical :: initial = .false.
        integer :: n_in = 0
        integer :: n_out = 0
        integer :: n_tot = 0
        type(prt_spec_t), dimension(:), allocatable :: prt_in
        type(prt_spec_t), dimension(:), allocatable :: prt_out
        type(string_t) :: method
        type(string_t) :: description
        class(prc_core_def_t), allocatable :: core_def
        character(32) :: md5sum = ""
    contains
        (Process libraries: process component def: TBP)
    end type process_component_def_t

```

Display the complete content.

```

(Process libraries: process component def: TBP)≡
    procedure :: write => process_component_def_write

```

*(Process libraries: procedures)*+≡

```

subroutine process_component_def_write (object, unit)
  class(process_component_def_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit)
  write (u, "(3x,A,A)") "Component ID      = ", char (object%basename)
  write (u, "(3x,A,L1)") "Initial component = ", object%initial
  write (u, "(3x,A,I0,1x,I0,1x,I0)") "N (in, out, tot) = ", &
    object%n_in, object%n_out, object%n_tot
  write (u, "(3x,A)", advance="no") "Particle content = "
  if (allocated (object%prt_in)) then
    call prt_spec_write (object%prt_in, u, advance="no")
  else
    write (u, "(A)", advance="no") "[undefined]"
  end if
  write (u, "(A)", advance="no") " => "
  if (allocated (object%prt_out)) then
    call prt_spec_write (object%prt_out, u, advance="no")
  else
    write (u, "(A)", advance="no") "[undefined]"
  end if
  write (u, "(A)")
  if (object%method /= "") then
    write (u, "(3x,A,A)") "Method          = ", &
      char (object%method)
  else
    write (u, "(3x,A)") "Method          = [undefined]"
  end if
  if (allocated (object%core_def)) then
    write (u, "(3x,A,A)") "Process variant = ", &
      char (object%core_def%type_string ())
    call object%core_def%write (u)
  else
    write (u, "(3x,A)") "Process variant = [undefined]"
  end if
  write (u, "(3x,A,A,A)") "MD5 sum (def)   = '", object%md5sum, "'"
end subroutine process_component_def_write

```

Read the process component definition. Allocate the process variant definition with appropriate type, matching the type name on file with the provided templates.

*(Process libraries: process component def: TBP)*+≡

```

procedure :: read => process_component_def_read

```

*(Process libraries: procedures)*+≡

```

subroutine process_component_def_read (component, unit, core_def_templates)
  class(process_component_def_t), intent(out) :: component
  integer, intent(in) :: unit
  type(prc_template_t), dimension(:), intent(in) :: core_def_templates
  character(80) :: buffer
  type(string_t) :: var_buffer, prefix, in_state, out_state
  type(string_t) :: variant_type

```

```

read (unit, "(A)")  buffer
call strip_prefix (buffer)
component%basename = trim (adjustl (buffer))

read (unit, "(A)")  buffer
call strip_prefix (buffer)
read (buffer, *)  component%initial

read (unit, "(A)")  buffer
call strip_prefix (buffer)
read (buffer, *)  component%n_in, component%n_out, component%n_tot

call get (unit, var_buffer)
call split (var_buffer, prefix, "=")  ! keeps 'in => out'
call split (var_buffer, prefix, "=")  ! actually: separator is '=>'

in_state = prefix
if (component%n_in > 0) then
  call prt_spec_read (component%prt_in, in_state)
end if

out_state = extract (var_buffer, 2)
if (component%n_out > 0) then
  call prt_spec_read (component%prt_out, out_state)
end if

read (unit, "(A)")  buffer
call strip_prefix (buffer)
component%method = trim (adjustl (buffer))
if (component%method == "[undefined]") &
  component%method = ""

read (unit, "(A)")  buffer
call strip_prefix (buffer)
variant_type = trim (adjustl (buffer))
call allocate_core_def &
  (core_def_templates, variant_type, component%core_def)
if (allocated (component%core_def)) then
  call component%core_def%read (unit)
end if

read (unit, "(A)")  buffer
call strip_prefix (buffer)
read (buffer(3:34), "(A32)")  component%md5sum

end subroutine process_component_def_read

```

Compute the MD5 sum of a process component. We reset the stored MD5 sum to the empty string (so a previous value is not included in the calculation), the write a temporary file and calculate the MD5 sum of that file.

This implies that all data that are displayed by the `write` method become part of the MD5 sum calculation.

The `model` is not part of the object, but must be included in the MD5 sum.

Otherwise, modifying the model and nothing else would not trigger remaking the process-component source. Note that the model parameters may change later and therefore are not incorporated.

After the MD5 sum of the component has been computed, we communicate it to the `writer` subobject of the specific `core_def` component. Although these types are abstract, the MD5-related features are valid for the abstract types.

```

(Process libraries: process component def: TBP)+≡
  procedure :: compute_md5sum => process_component_def_compute_md5sum

(Process libraries: procedures)+≡
  subroutine process_component_def_compute_md5sum (component, model)
    class(process_component_def_t), intent(inout) :: component
    type(model_t), intent(in), pointer :: model
    integer :: u
    component%md5sum = ""
    u = free_unit ()
    open (u, status = "scratch", action = "readwrite")
    if (associated (model)) write (u, "(A32)") model_get_md5sum (model)
    call component%write (u)
    rewind (u)
    component%md5sum = md5sum (u)
    close (u)
    if (allocated (component%core_def)) then
      call component%core_def%set_md5sum (component%md5sum)
    end if
  end subroutine process_component_def_compute_md5sum

```

Allocate the process driver (with a suitable type) for a process component. For internal processes, we may set all data already at this stage.

```

(Process libraries: process component def: TBP)+≡
  procedure :: allocate_driver => process_component_def_allocate_driver

(Process libraries: procedures)+≡
  subroutine process_component_def_allocate_driver (component, driver)
    class(process_component_def_t), intent(in) :: component
    class(prc_core_driver_t), intent(out), allocatable :: driver
    if (allocated (component%core_def)) then
      call component%core_def%allocate_driver (driver, component%basename)
    end if
  end subroutine process_component_def_allocate_driver

```

Tell whether the process core needs external code.

```

(Process libraries: process component def: TBP)+≡
  procedure :: needs_code => process_component_def_needs_code

(Process libraries: procedures)+≡
  function process_component_def_needs_code (component) result (flag)
    class(process_component_def_t), intent(in) :: component
    logical :: flag
    flag = component%core_def%needs_code ()
  end function process_component_def_needs_code

```

If there is external code, the `core_def` subobject should provide a writer object. This method returns a pointer to the writer.

```
<Process libraries: process component def: TBP>+≡
  procedure :: get_writer_ptr => process_component_def_get_writer_ptr

<Process libraries: procedures>+≡
  function process_component_def_get_writer_ptr (component) result (writer)
    class(process_component_def_t), intent(in), target :: component
    class(prc_writer_t), pointer :: writer
    writer => component%core_def%writer
  end function process_component_def_get_writer_ptr
```

Return an array which holds the names of all C functions that this process component implements.

```
<Process libraries: process component def: TBP>+≡
  procedure :: get_features => process_component_def_get_features

<Process libraries: procedures>+≡
  function process_component_def_get_features (component) result (features)
    class(process_component_def_t), intent(in) :: component
    type(string_t), dimension(:), allocatable :: features
    call component%core_def%get_features (features)
  end function process_component_def_get_features
```

Assign procedure pointers in the driver component (external processes). For internal processes, this is meaningless.

```
<Process libraries: process component def: TBP>+≡
  procedure :: connect => process_component_def_connect

<Process libraries: procedures>+≡
  subroutine process_component_def_connect &
    (component, lib_driver, i, proc_driver)
    class(process_component_def_t), intent(in) :: component
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
    select type (proc_driver)
    class is (process_driver_internal_t)
      ! nothing to do
    class default
      call component%core_def%connect (lib_driver, i, proc_driver)
    end select
  end subroutine process_component_def_connect
```

Return a pointer to the process core definition, which is of abstract type.

```
<Process libraries: process component def: TBP>+≡
  procedure :: get_core_def_ptr => process_component_get_core_def_ptr

<Process libraries: procedures>+≡
  function process_component_get_core_def_ptr (component) result (ptr)
    class(process_component_def_t), intent(in), target :: component
    class(prc_core_def_t), pointer :: ptr
    ptr => component%core_def
  end function process_component_get_core_def_ptr
```

Return nominal particle counts, as input by the user.

```

(Process libraries: process component def: TBP)+≡
  procedure :: get_n_in => process_component_def_get_n_in
  procedure :: get_n_out => process_component_def_get_n_out
  procedure :: get_n_tot => process_component_def_get_n_tot

(Process libraries: procedures)+≡
  function process_component_def_get_n_in (component) result (n_in)
    class(process_component_def_t), intent(in) :: component
    integer :: n_in
    n_in = component%n_in
  end function process_component_def_get_n_in

  function process_component_def_get_n_out (component) result (n_out)
    class(process_component_def_t), intent(in) :: component
    integer :: n_out
    n_out = component%n_out
  end function process_component_def_get_n_out

  function process_component_def_get_n_tot (component) result (n_tot)
    class(process_component_def_t), intent(in) :: component
    integer :: n_tot
    n_tot = component%n_tot
  end function process_component_def_get_n_tot

```

Return the MD5 sum.

```

(Process libraries: process component def: TBP)+≡
  procedure :: get_md5sum => process_component_def_get_md5sum

(Process libraries: procedures)+≡
  function process_component_def_get_md5sum (component) result (md5sum)
    class(process_component_def_t), intent(in) :: component
    character(32) :: md5sum
    md5sum = component%md5sum
  end function process_component_def_get_md5sum

```

## Process definition

The process component definitions are collected in a common process definition object.

The `id` is the ID string that the user has provided for identifying this process. It must be a string that is allowed as part of a Fortran variable name, since it may be used for generating code.

The number `n_in` is 1 or 2 for a decay or scattering process, respectively. This must be identical to `n_in` for all components.

The initial and extra component definitions (see above) are allocated as the `initial` and `extra` arrays, respectively. The latter are determined from the former.

The `md5sum` is used to verify the integrity of the configuration.

```

(Process libraries: public)+≡
  public :: process_def_t

```



```

<Process libraries: types>+≡
  type :: process_def_t
    private
      type(string_t) :: id
      type(model_t), pointer :: model => null ()
      type(string_t) :: model_name
      integer :: n_in = 0
      integer :: n_initial = 0
      integer :: n_extra = 0
      type(process_component_def_t), dimension(:), allocatable :: initial
      type(process_component_def_t), dimension(:), allocatable :: extra
      character(32) :: md5sum = ""
    contains
      <Process libraries: process def: TBP>
    end type process_def_t

```

Write the process definition including components:

```

<Process libraries: process def: TBP>≡
  procedure :: write => process_def_write

<Process libraries: procedures>+≡
  subroutine process_def_write (object, unit)
    class(process_def_t), intent(in) :: object
    integer, intent(in) :: unit
    integer :: i
    write (unit, "(1x,A,A,A)") "ID = ", char (object%id), ""
    select case (object%n_in)
      case (1); write (unit, "(1x,A)") "Decay"
      case (2); write (unit, "(1x,A)") "Scattering"
      case default
        write (unit, "(1x,A)") "[Undefined process]"
        return
    end select
    if (object%model_name /= "") then
      write (unit, "(1x,A,A)") "Model = ", char (object%model_name)
    else
      write (unit, "(1x,A)") "Model = [undefined]"
    end if
    write (unit, "(1x,A,I0)") "Initially defined component(s) = ", &
      object%n_initial
    write (unit, "(1x,A,I0)") "Extra generated component(s) = ", &
      object%n_extra
    write (unit, "(1x,A,A,A)") "MD5 sum = ", object%md5sum, ""
    if (allocated (object%initial)) then
      do i = 1, size (object%initial)
        write (unit, "(1x,A,I0)") "Component #", i
        call object%initial(i)%write (unit)
      end do
    end if
    if (allocated (object%extra)) then
      do i = 1, size (object%extra)
        write (unit, "(1x,A,I0)") "Component #", object%n_initial + i
        call object%extra(i)%write (unit)
      end do
    end if
  end subroutine process_def_write

```

```

        end if
    end subroutine process_def_write

```

Read the process definition including components.

*(Process libraries: process def: TBP)+≡*

```

    procedure :: read => process_def_read

```

*(Procedures: procedures)+≡*

```

subroutine process_def_read (object, unit, core_def_templates)
    class(process_def_t), intent(out) :: object
    integer, intent(in) :: unit
    type(prc_template_t), dimension(:), intent(in) :: core_def_templates
    integer :: i, i1, i2
    character(80) :: buffer, ref
    read (unit, "(A)") buffer
    call strip_prefix (buffer)
    i1 = scan (buffer, "'")
    i2 = scan (buffer, "'", back=.true.)
    if (i2 > i1) then
        object%id = buffer(i1+1:i2-1)
    else
        object%id = ""
    end if

    read (unit, "(A)") buffer
    select case (buffer(2:11))
    case ("Decay      "); object%n_in = 1
    case ("Scattering"); object%n_in = 2
    case default
        return
    end select

    read (unit, "(A)") buffer
    call strip_prefix (buffer)
    object%model_name = trim (adjustl (buffer))
    if (object%model_name == "[undefined]") object%model_name = ""

    read (unit, "(A)") buffer
    call strip_prefix (buffer)
    read (buffer, *) object%n_initial

    read (unit, "(A)") buffer
    call strip_prefix (buffer)
    read (buffer, *) object%n_extra

    read (unit, "(A)") buffer
    call strip_prefix (buffer)
    read (buffer(3:34), "(A32)") object%md5sum

    if (object%n_initial > 0) then
        allocate (object%initial (object%n_initial))
        do i = 1, object%n_initial
            read (unit, "(A)") buffer
            write (ref, "(1x,A,I0)") "Component #", i

```

```

        if (buffer /= ref) return ! Wrong component header
        call object%initial(i)%read (unit, core_def_templates)
    end do
end if

end subroutine process_def_read

```

Initialize an entry (initialize the process definition inside). We allocate the 'initial' set of components. Extra components remain unallocated.

The model should be present as a pointer. This allows us to retrieve the model's MD5 sum. However, for various tests it is sufficient to have the name.

We create the basenames for the process components by appending a suffix which we increment for each component.

```

<Process libraries: process def: TBP>+≡
    procedure :: init => process_def_init

<Process libraries: procedures>+≡
    subroutine process_def_init (def, id, model, model_name, n_in, n_components)
        class(process_def_t), intent(out) :: def
        type(string_t), intent(in), optional :: id
        type(model_t), intent(in), optional, target :: model
        type(string_t), intent(in), optional :: model_name
        integer, intent(in), optional :: n_in
        integer, intent(in), optional :: n_components
        character(16) :: suffix
        integer :: i
        if (present (id)) then
            def%id = id
        else
            def%id = ""
        end if
        if (present (model)) then
            def%model => model
            def%model_name = model%get_name ()
        else
            def%model => null ()
            if (present (model_name)) then
                def%model_name = model_name
            else
                def%model_name = ""
            end if
        end if
        if (present (n_in)) def%n_in = n_in
        if (present (n_components)) then
            def%n_initial = n_components
            allocate (def%initial (n_components))
        end if
        def%initial%initial = .true.
        def%initial%method = ""
        do i = 1, def%n_initial
            write (suffix, "(A,I0)") "_i", i
            def%initial(i)%basename = def%id // trim (suffix)
        end do
        def%initial%description = ""
    end subroutine process_def_init

```

```
end subroutine process_def_init
```

Initialize an initial component. The particle content must be specified. The process core block is not (yet) allocated.

We assume that the particle arrays match the `n_in` and `n_out` values in size. The model is referred to by name; it is identified as an existing model later. The index `i` must refer to an existing element of the component array.

Data specific for the process core of a component are imported as the `core_def` argument. We should allocate an object of class `prc_core_def_t` with the appropriate specific type, fill it, and transfer it to the process component definition here. The allocation is moved, so the original allocated object is returned empty.

*(Process libraries: process def: TBP)+≡*

```
procedure :: import_component => process_def_import_component
```

*(Process libraries: procedures)+≡*

```
subroutine process_def_import_component (def, &
    i, n_out, prt_in, prt_out, method, variant)
class(process_def_t), intent(inout) :: def
integer, intent(in) :: i
integer, intent(in), optional :: n_out
type(prt_spec_t), dimension(:), intent(in), optional :: prt_in
type(prt_spec_t), dimension(:), intent(in), optional :: prt_out
type(string_t), intent(in), optional :: method
class(prc_core_def_t), &
    intent(inout), allocatable, optional :: variant
integer :: p
associate (comp => def%initial(i))
    if (present (n_out)) then
        comp%n_in = def%n_in
        comp%n_out = n_out
        comp%n_tot = def%n_in + n_out
    end if
    if (present (prt_in)) then
        allocate (comp%prt_in (size (prt_in)))
        comp%prt_in = prt_in
    end if
    if (present (prt_out)) then
        allocate (comp%prt_out (size (prt_out)))
        comp%prt_out = prt_out
    end if
    if (present (method)) comp%method = method
    if (present (variant)) then
        call move_alloc (variant, comp%core_def)
    end if
    if (allocated (comp%prt_in) .and. allocated (comp%prt_out)) then
        associate (d => comp%description)
            d = ""
            do p = 1, size (prt_in)
                if (p > 1) d = d // ", "
                d = d // comp%prt_in(p)%to_string ()
            end do
            d = d // " => "
        end associate
    end if
end associate
end subroutine
```

```

do p = 1, size (prt_out)
  if (p > 1) d = d // ", "
  d = d // comp%prt_out(p)%to_string ()
end do
if (comp%method /= "") then
  d = d // " [" // comp%method // "]"
end if
end associate
end if
end associate
end subroutine process_def_import_component

```

Compute the MD5 sum for this process definition. We compute the MD5 sums for all components individually, then concatenate a string of those and compute the MD5 sum of this string. We also include the model name. All other data part of the component definitions.

```

<Process libraries: process def: TBP>+≡
  procedure :: compute_md5sum => process_def_compute_md5sum

<Process libraries: procedures>+≡
  subroutine process_def_compute_md5sum (def)
    class(process_def_t), intent(inout) :: def
    integer :: i
    type(string_t) :: buffer
    buffer = def%model_name
    do i = 1, def%n_initial
      call def%initial(i)%compute_md5sum (def%model)
      buffer = buffer // def%initial(i)%md5sum
    end do
    do i = 1, def%n_extra
      call def%extra(i)%compute_md5sum (def%model)
      buffer = buffer // def%initial(i)%md5sum
    end do
    def%md5sum = md5sum (char (buffer))
  end subroutine process_def_compute_md5sum

```

Return the MD5 sum of the process or of a process component.

```

<Process libraries: process def: TBP>+≡
  procedure :: get_md5sum => process_def_get_md5sum

<Process libraries: procedures>+≡
  function process_def_get_md5sum (def, i_component) result (md5sum)
    class(process_def_t), intent(in) :: def
    integer, intent(in), optional :: i_component
    character(32) :: md5sum
    if (present (i_component)) then
      md5sum = def%initial(i_component)%md5sum
    else
      md5sum = def%md5sum
    end if
  end function process_def_get_md5sum

```

This query tells whether a specific process components relies on external code. This includes all traditional WHIZARD matrix elements which rely on O'MEGA for code generation. Other process components (trivial decays, subtraction terms) do not require external code.

NOTE: Implemented only for initial component.

The query is passed to the process component.

```

<Process libraries: process def: TBP>+≡
  procedure :: needs_code => process_def_needs_code

<Process libraries: procedures>+≡
  function process_def_needs_code (def, i_component) result (flag)
    class(process_def_t), intent(in) :: def
    integer, intent(in) :: i_component
    logical :: flag
    flag = def%initial(i_component)%needs_code ()
  end function process_def_needs_code

```

### Process definition list

A list of process definitions is the starting point for creating a process library. The list is built when reading the user input. When reading an existing process library, the list is used for cross-checking and updating the configuration.

We need a type for the list entry. The simplest way is to extend the process definition type, so all methods apply to the process definition directly.

```

<Process libraries: public>+≡
  public :: process_def_entry_t

<Process libraries: types>+≡
  type, extends (process_def_t) :: process_def_entry_t
    private
    type(process_def_entry_t), pointer :: next => null ()
  end type process_def_entry_t

```

This is the type for the list itself.

```

<Process libraries: types>+≡
  type :: process_def_list_t
    private
    type(process_def_entry_t), pointer :: first => null ()
    type(process_def_entry_t), pointer :: last => null ()
  contains
    <Process libraries: process def list: TBP>
  end type process_def_list_t

```

The deallocates the list iteratively. We assume that the list entries do not need finalization themselves.

```

<Process libraries: process def list: TBP>≡
  procedure :: final => process_def_list_final

```

```

<Process libraries: procedures>+≡
subroutine process_def_list_final (list)
  class(process_def_list_t), intent(inout) :: list
  type(process_def_entry_t), pointer :: current
  nullify (list%last)
  do while (associated (list%first))
    current => list%first
    list%first => current%next
    deallocate (current)
  end do
end subroutine process_def_list_final

```

Write the complete list.

```

<Process libraries: process def list: TBP>+≡
  procedure :: write => process_def_list_write

<Process libraries: procedures>+≡
subroutine process_def_list_write (object, unit)
  class(process_def_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(process_def_entry_t), pointer :: entry
  integer :: i, u
  u = output_unit (unit)
  if (associated (object%first)) then
    i = 1
    entry => object%first
    do while (associated (entry))
      write (u, "(1x,A,IO,A)") "Process #", i, ":"
      call entry%write (u)
      i = i + 1
      entry => entry%next
      if (associated (entry)) write (u, *)
    end do
  else
    write (u, "(1x,A)") "Process definition list: [empty]"
  end if
end subroutine process_def_list_write

```

Read the complete list. We need an array of templates for the component sub-objects of abstract `prc_core_t` type, to allocate them with the correct specific type.

NOTE: Error handling is missing. Reading will just be aborted on error, or an I/O error occurs.

```

<Process libraries: process def list: TBP>+≡
  procedure :: read => process_def_list_read

<Process libraries: procedures>+≡
subroutine process_def_list_read (object, unit, core_def_templates)
  class(process_def_list_t), intent(out) :: object
  integer, intent(in) :: unit
  type(prc_template_t), dimension(:), intent(in) :: core_def_templates
  type(process_def_entry_t), pointer :: entry
  character(80) :: buffer, ref

```

```

integer :: i
read (unit, "(A)")  buffer
write (ref, "(1x,A)") "Process definition list: [empty]"
if (buffer == ref) return      ! OK: empty library
backspace (unit)
READ_ENTRIES: do i = 1, huge (0)
  if (i > 1) read (unit, *, end=1)
  read (unit, "(A)")  buffer

  write (ref, "(1x,A,I0,A)") "Process #", i, ":"
  if (buffer /= ref) return    ! Wrong process header: done.
  allocate (entry)
  call entry%read (unit, core_def_templates)
  call object%append (entry)
end do READ_ENTRIES
1  continue                ! EOF: done
end subroutine process_def_list_read

```

Append an entry to the list. The entry should be allocated as a pointer, and the pointer allocation is transferred. The original pointer is returned null.

```

<Process libraries: process def list: TBP>+≡
  procedure :: append => process_def_list_append

<Process libraries: procedures>+≡
  subroutine process_def_list_append (list, entry)
    class(process_def_list_t), intent(inout) :: list
    type(process_def_entry_t), intent(inout), pointer :: entry
    if (associated (list%first)) then
      list%last%next => entry
    else
      list%first => entry
    end if
    list%last => entry
    entry => null ()
  end subroutine process_def_list_append

```

### Probe the process definition list

Return the number of processes supported by the library.

```

<Process libraries: process def list: TBP>+≡
  procedure :: get_n_processes => process_def_list_get_n_processes

<Process libraries: procedures>+≡
  function process_def_list_get_n_processes (list) result (n)
    integer :: n
    class(process_def_list_t), intent(in) :: list
    type(process_def_entry_t), pointer :: current
    n = 0
    current => list%first
    do while (associated (current))
      n = n + 1
      current => current%next
    end do
  end function

```



```
end function process_def_list_get_n_processes
```

Allocate an array with the process IDs supported by the library.

```
<Process libraries: process def list: TBP>+≡
  procedure :: get_process_id_list => process_def_list_get_process_id_list

<Process libraries: procedures>+≡
  subroutine process_def_list_get_process_id_list (list, id)
    class(process_def_list_t), intent(in) :: list
    type(string_t), dimension(:), allocatable, intent(out) :: id
    type(process_def_entry_t), pointer :: current
    integer :: i
    allocate (id (list%get_n_processes ()))
    i = 0
    current => list%first
    do while (associated (current))
      i = i + 1
      id(i) = current%id
      current => current%next
    end do
  end subroutine process_def_list_get_process_id_list
```

Return true if a given process is in the library.

```
<Process libraries: process def list: TBP>+≡
  procedure :: contains => process_def_list_contains

<Process libraries: procedures>+≡
  function process_def_list_contains (list, id) result (flag)
    logical :: flag
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    type(process_def_entry_t), pointer :: current
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        flag = .true.; return
      end if
      current => current%next
    end do
    flag = .false.
  end function process_def_list_contains
```

Return the index of the entry that corresponds to a given process.

```
<Process libraries: process def list: TBP>+≡
  procedure :: get_entry_index => process_def_list_get_entry_index

<Process libraries: procedures>+≡
  function process_def_list_get_entry_index (list, id) result (n)
    integer :: n
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    type(process_def_entry_t), pointer :: current
    n = 0
    current => list%first
```

```

do while (associated (current))
  n = n + 1
  if (id == current%id) then
    return
  end if
  current => current%next
end do
n = 0
end function process_def_list_get_entry_index

```

Return the model name for a given process in the library.

*(Process libraries: process def list: TBP)+≡*

```

procedure :: get_model_name => process_def_list_get_model_name

```

*(Process libraries: procedures)+≡*

```

function process_def_list_get_model_name (list, id) result (model_name)
  type(string_t) :: model_name
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
    if (id == current%id) then
      model_name = current%model_name
      return
    end if
    current => current%next
  end do
  model_name = ""
end function process_def_list_get_model_name

```

Return the number of incoming particles of a given process in the library. This tells us whether the process is a decay or a scattering.

*(Process libraries: process def list: TBP)+≡*

```

procedure :: get_n_in => process_def_list_get_n_in

```

*(Process libraries: procedures)+≡*

```

function process_def_list_get_n_in (list, id) result (n)
  integer :: n
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
    if (id == current%id) then
      n = current%n_in
      return
    end if
    current => current%next
  end do
end function process_def_list_get_n_in

```

Return the number of components of a given process in the library.

```

(Process libraries: process def list: TBP)+≡
  procedure :: get_n_components => process_def_list_get_n_components

(Process libraries: procedures)+≡
  function process_def_list_get_n_components (list, id) result (n)
    integer :: n
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    type(process_def_entry_t), pointer :: current
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        n = current%n_initial + current%n_extra
        return
      end if
      current => current%next
    end do
  end function process_def_list_get_n_components

```

Return a pointer to a specific process component definition.

```

(Process libraries: process def list: TBP)+≡
  procedure :: get_component_def_ptr => process_def_list_get_component_def_ptr

(Process libraries: procedures)+≡
  function process_def_list_get_component_def_ptr (list, id, i) result (ptr)
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    integer, intent(in) :: i
    type(process_component_def_t), pointer :: ptr
    type(process_def_entry_t), pointer :: current
    ptr => null ()
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        if (i <= current%n_initial) then
          ptr => current%initial(i)
        else if (i <= current%n_initial + current%n_extra) then
          ptr => current%extra(i-current%n_initial)
        end if
        return
      end if
      current => current%next
    end do
  end function process_def_list_get_component_def_ptr

```

Return the list of component IDs of a given process in the library.

```

(Process libraries: process def list: TBP)+≡
  procedure :: get_component_list => process_def_list_get_component_list

(Process libraries: procedures)+≡
  subroutine process_def_list_get_component_list (list, id, cid)
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id

```

```

type(string_t), dimension(:), allocatable, intent(out) :: cid
type(process_def_entry_t), pointer :: current
integer :: i, n
current => list%first
do while (associated (current))
  if (id == current%id) then
    allocate (cid (current%n_initial + current%n_extra))
    do i = 1, current%n_initial
      cid(i) = current%initial(i)%basename
    end do
    n = current%n_initial
    do i = 1, current%n_extra
      cid(n + i) = current%extra(i)%basename
    end do
    return
  end if
  current => current%next
end do
end subroutine process_def_list_get_component_list

```

Return the list of component description strings for a given process in the library.

*(Process libraries: process def list: TBP)+≡*

```

procedure :: get_component_description_list => &
  process_def_list_get_component_description_list

```

*(Procedures: procedures)+≡*

```

subroutine process_def_list_get_component_description_list &
  (list, id, description)
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(string_t), dimension(:), allocatable, intent(out) :: description
  type(process_def_entry_t), pointer :: current
  integer :: i, n
  current => list%first
  do while (associated (current))
    if (id == current%id) then
      allocate (description (current%n_initial + current%n_extra))
      do i = 1, current%n_initial
        description(i) = current%initial(i)%description
      end do
      n = current%n_initial
      do i = 1, current%n_extra
        description(n + i) = current%extra(i)%description
      end do
      return
    end if
    current => current%next
  end do
end subroutine process_def_list_get_component_description_list

```

### 14.4.3 Process library

The process library object is the interface between the process definition data, as provided by the user, generated or linked process code on file, and the process run data that reference the process code.

#### Process library entry

For each process component that is part of the library, there is a separate library entry (`process_library_entry_t`). The library entry connects a process definition with the specific code (if any) in the compiled driver library.

The `status` indicates how far the process has been processed by the system (definition, code generation, compilation, linking). A process with status `STAT_LOADED` is accessible for computing matrix elements.

The `def` pointer identifies the corresponding process definition. The process component within that definition is identified by the `i_component` index.

The `i_external` index refers to the compiled library driver. If it is zero, there is no associated matrix-element code.

The `driver` component holds the pointers to the matrix-element specific functions, in particular the matrix element function itself.

```
<Process libraries: types>+≡
type :: process_library_entry_t
  private
  integer :: status = STAT_UNKNOWN
  type(process_def_t), pointer :: def => null ()
  integer :: i_component = 0
  integer :: i_external = 0
  class(prc_core_driver_t), allocatable :: driver
contains
  <Process libraries: process library entry: TBP>
end type process_library_entry_t
```

Here are the available status codes. An entry starts with `UNKNOWN` status. Once the association with a valid process definition is established, the status becomes `CONFIGURED`. If matrix element source code is to be generated by the system or provided from elsewhere, `CODE_GENERATED` indicates that this is done. The `COMPILED` status is next, it also applies to processes which are accessed as pre-compiled binaries. Finally, the library is linked and process pointers are set; this is marked as `LOADED`.

For a process library, the initial status is `OPEN`, since process definitions may be added. After configuration, the process content is fixed and the status becomes `CONFIGURED`. The further states are as above, always referring to the lowest status among the process entries.

```
<Process libraries: parameters>≡
integer, parameter :: STAT_UNKNOWN = 0
integer, parameter :: STAT_OPEN = 1
integer, parameter :: STAT_CONFIGURED = 2
integer, parameter :: STAT_SOURCE = 3
integer, parameter :: STAT_COMPILED = 4
integer, parameter :: STAT_LINKED = 5
integer, parameter :: STAT_ACTIVE = 6
```

These are the associated code letters, for output:

```

(Process libraries: parameters)+≡
    character, dimension(0:6), parameter :: STATUS_LETTER = &
        ["?", "o", "f", "s", "c", "l", "a"]

```

This produces a condensed account of the library entry. The status is indicated by a letter in brackets, then the ID and component index of the associated process definition, finally the library index, if available.

```

(Process libraries: process library entry: TBP)≡
    procedure :: to_string => process_library_entry_to_string

(Process libraries: procedures)+≡
    function process_library_entry_to_string (object) result (string)
        type(string_t) :: string
        class(process_library_entry_t), intent(in) :: object
        character(32) :: buffer
        string = "[" // STATUS_LETTER(object%status) // "]"
        select case (object%status)
        case (STAT_UNKNOWN)
        case default
            if (associated (object%def)) then
                write (buffer, "(IO)") object%i_component
                string = string // " " // object%def%id // "." // trim (buffer)
            end if
            if (object%i_external /= 0) then
                write (buffer, "(IO)") object%i_external
                string = string // " = ext:" // trim (buffer)
            else
                string = string // " = int"
            end if
            if (allocated (object%driver)) then
                string = string // " (" // object%driver%type_name () // ")"
            end if
        end select
    end function process_library_entry_to_string

```

Initialize with data. Used for the unit tests.

```

(Process libraries: process library entry: TBP)+≡
    procedure :: init => process_library_entry_init

(Process libraries: procedures)+≡
    subroutine process_library_entry_init (object, &
        status, def, i_component, i_external)
        class(process_library_entry_t), intent(out) :: object
        integer, intent(in) :: status
        type(process_def_t), target, intent(in) :: def
        integer, intent(in) :: i_component
        integer, intent(in) :: i_external
        object%status = status
        object%def => def
        object%i_component = i_component
        object%i_external = i_external
    end subroutine process_library_entry_init

```

Assign pointers for all process-specific features. We have to combine the method from the `core_def` specification, the assigned pointers within the library driver, the index within that driver, and the process driver which should receive the links.

```

(Process libraries: process library entry: TBP)+≡
  procedure :: connect => process_library_entry_connect

(Process libraries: procedures)+≡
  subroutine process_library_entry_connect (entry, lib_driver, i)
    class(process_library_entry_t), intent(inout) :: entry
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    call entry%def%initial(entry%i_component)%connect &
      (lib_driver, i, entry%driver)
  end subroutine process_library_entry_connect

```

## The process library object

The `process_library_t` type is an extension of the `process_def_list_t` type. Thus, it automatically contains the process definition list.

The `basename` identifies the library generically.

The `external` flag is true if any process within the library needs external code, so the library must correspond to an actual code library (statically or dynamically linked).

The `entry` array contains all process components that can be handled by this library. Each entry refers to the process (component) definition and to the associated external matrix element code, if there is any.

The `driver` object is needed only if `external` is true. This object handles all interactions with external matrix-element code.

The `md5sum` summarizes the complete `process_def_list_t` base object. It can be used to check if the library configuration has changed.

```

(Process libraries: public)+≡
  public :: process_library_t

(Process libraries: types)+≡
  type, extends (process_def_list_t) :: process_library_t
    private
    type(string_t) :: basename
    integer :: n_entries = 0
    logical :: external = .false.
    integer :: status = STAT_UNKNOWN
    logical :: driver_exists = .false.
    logical :: makefile_exists = .false.
    type(process_library_entry_t), dimension(:), allocatable :: entry
    type(prclib_driver_t) :: driver
    character(32) :: md5sum = ""
  contains
    (Process libraries: process library: TBP)
  end type process_library_t

```

For the output, we write first the metadata and the DL access record, then the library entries in short form, and finally the process definition list which is the base object.

```

(Process libraries: process library: TBP)≡
  procedure :: write => process_library_write

(Process libraries: procedures)+≡
  subroutine process_library_write (object, unit)
    class(process_library_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: i, u
    u = output_unit (unit)
    write (u, "(1x,A,A)") "Process library: ", char (object%basename)
    write (u, "(3x,A,L1)") "external      = ", object%external
    write (u, "(3x,A,L1)") "makefile exists = ", object%makefile_exists
    write (u, "(3x,A,L1)") "driver exists  = ", object%driver_exists
    write (u, "(3x,A,A1)") "code status   = ", &
      STATUS_LETTER (object%status)
    !   write (u, "(3x,A,A,A)") "MD5 sum      = '", object%md5sum, "'"
    write (u, *)
    if (allocated (object%entry)) then
      write (u, "(1x,A)", advance="no") "Process library entries:"
      write (u, "(1x,I0)") object%n_entries
      do i = 1, size (object%entry)
        write (u, "(1x,A,I0,A,A)") "Entry #", i, ": ", &
          char (object%entry(i)%to_string ())
      end do
      write (u, *)
    end if
    if (object%external) then
      call object%driver%write (u)
      write (u, *)
    end if
    call object%process_def_list_t%write (u)
  end subroutine process_library_write

```

The initializer defines just the basename. We may now add process definitions to the library.

```

(Process libraries: process library: TBP)+≡
  procedure :: init => process_library_init

(Process libraries: procedures)+≡
  subroutine process_library_init (lib, basename)
    class(process_library_t), intent(out) :: lib
    type(string_t), intent(in) :: basename
    lib%basename = basename
    lib%status = STAT_OPEN
    call msg_message ("Process library '" // char (basename) &
      // "' : initialized")
  end subroutine process_library_init

```

The `configure` procedure scans the allocated entries in the process definition list. The configuration proceeds in three passes.



In the first pass, we scan the process definition list and count the number of process components and the number of components which need external code. This is used to allocate the **entry** array.

In the second pass, we initialize the **entry** elements which connect process definitions, process driver objects, and external code.

In the third pass, we initialize the library driver object, allocating an entry for each external matrix element.

NOTE: Currently we handle only **initial** process components; **extra** components are ignored.

```

(Process libraries: process library: TBP)+≡
  procedure :: configure => process_library_configure

(Process libraries: procedures)+≡
  subroutine process_library_configure (lib)
    class(process_library_t), intent(inout) :: lib
    type(process_def_entry_t), pointer :: def_entry
    integer :: n_entries, n_external, i_entry, i_external
    type(string_t) :: model_name
    integer :: i_component

    n_entries = 0
    n_external = 0
    if (allocated (lib%entry)) deallocate (lib%entry)

    def_entry => lib%first
    do while (associated (def_entry))
      do i_component = 1, def_entry%n_initial
        n_entries = n_entries + 1
        if (def_entry%initial(i_component)%needs_code ()) then
          n_external = n_external + 1
          lib%external = .true.
        end if
      end do
      def_entry => def_entry%next
    end do
    lib%n_entries = n_entries

    allocate (lib%entry (n_entries))
    i_entry = 0
    i_external = 0
    def_entry => lib%first
    do while (associated (def_entry))
      do i_component = 1, def_entry%n_initial
        i_entry = i_entry + 1
        associate (lib_entry => lib%entry(i_entry))
          lib_entry%status = STAT_CONFIGURED
          lib_entry%def => def_entry%process_def_t
          lib_entry%i_component = i_component
          if (def_entry%initial(i_component)%needs_code ()) then
            i_external = i_external + 1
            lib_entry%i_external = i_external
          end if
          call def_entry%initial(i_component)%allocate_driver &
            (lib_entry%driver)
        end associate
      end do
      def_entry => def_entry%next
    end do
  end subroutine

```

```

        end associate
    end do
    def_entry => def_entry%next
end do

call lib%driver%init (lib%basename, n_external)
do i_entry = 1, n_entries
    associate (lib_entry => lib%entry(i_entry))
        i_component = lib_entry%i_component
        model_name = lib_entry%def%model_name
        associate (def => lib_entry%def%initial(i_component))
            if (def%needs_code ()) then
                call lib%driver%set_record (lib_entry%i_external, &
                    def%basename, &
                    model_name, &
                    def%get_features (), def%get_writer_ptr ())
            end if
        end associate
    end associate
end do

if (lib%external) then
    where (lib%entry%i_external == 0) lib%entry%status = STAT_LINKED
    lib%status = STAT_CONFIGURED
    lib%makefile_exists = .false.
    lib%driver_exists = .false.
else
    if (lib%n_entries /= 0) lib%entry%status = STAT_LINKED
    lib%status = STAT_LINKED
end if
end subroutine process_library_configure

```

Compute the MD5 sum. We concatenate the individual MD5 sums of all processes (which, in turn, are derived from the MD5 sums of their components) and compute the MD5 sum of that.

This should be executed *after* configuration, where the driver was initialized, since otherwise the MD5 sum stored in the driver would be overwritten.

*(Process libraries: process library: TBP)+≡*

```

    procedure :: compute_md5sum => process_library_compute_md5sum

```

*(Process libraries: procedures)+≡*

```

subroutine process_library_compute_md5sum (lib)
    class(process_library_t), intent(inout) :: lib
    type(process_def_entry_t), pointer :: def_entry
    type(string_t) :: buffer
    buffer = lib%basename
    def_entry => lib%first
    do while (associated (def_entry))
        call def_entry%compute_md5sum ()
        buffer = buffer // def_entry%md5sum
        def_entry => def_entry%next
    end do
    lib%md5sum = md5sum (char (buffer))
    call lib%driver%set_md5sum (lib%md5sum)

```

```
end subroutine process_library_compute_md5sum
```

Write an appropriate makefile, if there are external processes. Unless `force` is in effect, first check if there is already a makefile with the correct MD5 sum. If yes, do nothing.

```
<Process libraries: process library: TBP>+≡
  procedure :: write_makefile => process_library_write_makefile

<Process libraries: procedures>+≡
  subroutine process_library_write_makefile (lib, os_data, force)
    class(process_library_t), intent(inout) :: lib
    type(os_data_t), intent(in) :: os_data
    logical, intent(in) :: force
    character(32) :: md5sum_file
    logical :: generate
    integer :: unit
    if (lib%external) then
      generate = .true.
      if (.not. force) then
        md5sum_file = lib%driver%get_md5sum_makefile ()
        if (lib%md5sum == md5sum_file) then
          call msg_message ("Process library '" // char (lib%basename) &
            // "': keeping makefile")
          generate = .false.
        end if
      end if
    end if
    if (generate) then
      call msg_message ("Process library '" // char (lib%basename) &
        // "': writing makefile")
      unit = free_unit ()
      open (unit, file = char (lib%driver%basename // ".makefile"), &
        status="replace", action="write")
      call lib%driver%generate_makefile (unit, os_data)
      close (unit)
    end if
    lib%makefile_exists = .true.
  end if
end subroutine process_library_write_makefile
```

Write the driver source code for the library to file, if there are external processes.

```
<Process libraries: process library: TBP>+≡
  procedure :: write_driver => process_library_write_driver

<Process libraries: procedures>+≡
  subroutine process_library_write_driver (lib, force)
    class(process_library_t), intent(inout) :: lib
    logical, intent(in) :: force
    character(32) :: md5sum_file
    logical :: generate
    integer :: unit
    if (lib%external) then
      generate = .true.
      if (.not. force) then
        md5sum_file = lib%driver%get_md5sum_driver ()
```

```

        if (lib%md5sum == md5sum_file) then
            call msg_message ("Process library '" // char (lib%basename) &
                // "': keeping driver")
            generate = .false.
        end if
    end if
    if (generate) then
        call msg_message ("Process library '" // char (lib%basename) &
            // "': writing driver")
        unit = free_unit ()
        open (unit, file = char (lib%driver%basename // ".f90"), &
            status="replace", action="write")
        call lib%driver%generate_driver_code (unit)
        close (unit)
    end if
    lib%driver_exists = .true.
end if
end subroutine process_library_write_driver

```

Update the compilation status of an external library.

Strictly speaking, this is not necessary for a one-time run, since the individual library methods will update the status themselves. However, it allows us to identify compilation steps that we can skip because the file exists or is already loaded, for the whole library or for particular entries.

Independently, the building process is controlled by a makefile. Thus, previous files are reused if they are not modified by the current compilation.

1. If it is not already loaded, attempt to load the library. If successful, check the overall MD5 sum. If it matches, just keep it loaded and mark as ACTIVE. If not, check the MD5 sum for all linked process components. Where it matches, mark the entry as COMPILED. Then, unload the library and mark as CONFIGURED.

Thus, we can identify compiled files for all matrix elements which are accessible via the previous compiled library, even if it is no longer up to date.

2. If the library is now in CONFIGURED state, look for valid source files. Each entry that is just in CONFIGURED state will advance to SOURCE if the MD5 sum matches. Finally, advance the whole library to SOURCE if all entries are at least in this condition.

*(Process libraries: process library: TBP)+≡*

```

    procedure :: update_status => process_library_update_status

```

*(Process libraries: procedures)+≡*

```

    subroutine process_library_update_status (lib, os_data)
        class(process_library_t), intent(inout) :: lib
        type(os_data_t), intent(in) :: os_data
        character(32) :: md5sum_file
        type(process_component_def_t), pointer :: def
        integer :: i, i_external, i_component
        if (lib%external) then
            select case (lib%status)

```

```

case (STAT_CONFIGURED:STAT_LINKED)
    call lib%driver%load (os_data, noerror=.true.)
end select
if (lib%driver%loaded) then
    md5sum_file = lib%driver%get_md5sum (0)
    if (lib%md5sum == md5sum_file) then
        call lib%load_entries ()
        lib%entry%status = STAT_ACTIVE
        lib%status = STAT_ACTIVE
        call msg_message ("Process library '" // char (lib%basename) &
            // "': active")
    else
        do i = 1, lib%n_entries
            associate (entry => lib%entry(i))
                i_external = entry%i_external
                i_component = entry%i_component
                if (i_external /= 0) then
                    md5sum_file = lib%driver%get_md5sum (i_external)
                    if (entry%def%get_md5sum (i_component) == md5sum_file) then
                        entry%status = STAT_COMPILED
                    else
                        entry%status = STAT_CONFIGURED
                    end if
                end if
            end associate
        end do
        call lib%driver%unload ()
        lib%status = STAT_CONFIGURED
    end if
end if
select case (lib%status)
case (STAT_CONFIGURED)
    do i = 1, lib%n_entries
        associate (entry => lib%entry(i))
            i_external = entry%i_external
            i_component = entry%i_component
            if (i_external /= 0) then
                select case (entry%status)
                case (STAT_CONFIGURED)
                    md5sum_file = lib%driver%get_md5sum_source (i_external)
                    if (entry%def%get_md5sum (i_component) == md5sum_file) then
                        entry%status = STAT_SOURCE
                    end if
                end select
            end if
        end associate
    end do
    if (all (lib%entry%status >= STAT_SOURCE)) then
        md5sum_file = lib%driver%get_md5sum_driver ()
        if (lib%md5sum == md5sum_file) then
            lib%status = STAT_SOURCE
        end if
    end if
end select

```

```

        end if
    end subroutine process_library_update_status

```

This procedure triggers code generation for all processes where this is possible.

We generate code only for external processes of status `STAT_CONFIGURED`, which then advance to `STAT_SOURCE`. If, for a particular process, the status is already advanced, we do not remove previous files, so `make` will consider them as up to date if they exist. Otherwise, we remove those files to force a fresh `make`.

Finally, if any source code has been generated, we need a driver file.

```

<Process libraries: process library: TBP>+≡
    procedure :: make_source => process_library_make_source

<Process libraries: procedures>+≡
    subroutine process_library_make_source (lib, os_data)
        class(process_library_t), intent(inout) :: lib
        type(os_data_t), intent(in) :: os_data
        integer :: i, i_external
        if (lib%external) then
            select case (lib%status)
            case (STAT_CONFIGURED)
                call msg_message ("Process library '" // char (lib%basename) &
                    // "': creating source code")
                do i = 1, size (lib%entry)
                    associate (entry => lib%entry(i))
                        i_external = entry%i_external
                        if (i_external /= 0 &
                            .and. lib%entry(i)%status == STAT_CONFIGURED) then
                            call lib%driver%clean_proc (i_external, os_data)
                        end if
                    end associate
                end do
                call lib%driver%make_source (os_data)
                lib%status = STAT_SOURCE
                where (lib%entry%i_external /= 0 &
                    .and. lib%entry%status == STAT_CONFIGURED)
                    lib%entry%status = STAT_SOURCE
                end where
                lib%status = STAT_SOURCE
            end select
        end if
    end subroutine process_library_make_source

```

Compile the generated code and update the status codes. Try to make the sources first, just in case.

```

<Process libraries: process library: TBP>+≡
    procedure :: make_compile => process_library_make_compile

<Process libraries: procedures>+≡
    subroutine process_library_make_compile (lib, os_data)
        class(process_library_t), intent(inout) :: lib
        type(os_data_t), intent(in) :: os_data
        if (lib%external) then

```

```

select case (lib%status)
case (STAT_CONFIGURED)
    call lib%make_source (os_data)
end select
select case (lib%status)
case (STAT_SOURCE)
    call msg_message ("Process library '" // char (lib%basename) &
        // "': compiling sources")
    call lib%driver%make_compile (os_data)
    where (lib%entry%i_external /= 0 &
        .and. lib%entry%status == STAT_SOURCE)
        lib%entry%status = STAT_COMPILED
    end where
    lib%status = STAT_COMPILED
end select
end if
end subroutine process_library_make_compile

```

Link the process library. Try to compile first, just in case.

```

<Process libraries: process library: TBP>+≡
    procedure :: make_link => process_library_make_link

<Process libraries: procedures>+≡
    subroutine process_library_make_link (lib, os_data)
    class(process_library_t), intent(inout) :: lib
    type(os_data_t), intent(in) :: os_data
    if (lib%external) then
        select case (lib%status)
        case (STAT_CONFIGURED:STAT_SOURCE)
            call lib%make_compile (os_data)
        end select
        select case (lib%status)
        case (STAT_COMPILED)
            call msg_message ("Process library '" // char (lib%basename) &
                // "': linking")
            call lib%driver%make_link (os_data)
            lib%entry%status = STAT_LINKED
            lib%status = STAT_LINKED
        end select
    end if
    end subroutine process_library_make_link

```

Load the process library, i.e., assign pointers to the library functions.

```

<Process libraries: process library: TBP>+≡
    procedure :: load => process_library_load

<Process libraries: procedures>+≡
    subroutine process_library_load (lib, os_data)
    class(process_library_t), intent(inout) :: lib
    type(os_data_t), intent(in) :: os_data
    select case (lib%status)
    case (STAT_CONFIGURED:STAT_COMPILED)
        call lib%make_link (os_data)
    end select

```

```

select case (lib%status)
case (STAT_LINKED)
  if (lib%external) then
    call msg_message ("Process library '" // char (lib%basename) &
      // "': loading")
    call lib%driver%load (os_data)
    call lib%load_entries ()
  end if
  lib%entry%status = STAT_ACTIVE
  lib%status = STAT_ACTIVE
end select
end subroutine process_library_load

```

This is the actual loading part for the process methods.

```

<Process libraries: process library: TBP>+≡
  procedure :: load_entries => process_library_load_entries

<Process libraries: procedures>+≡
  subroutine process_library_load_entries (lib)
    class(process_library_t), intent(inout) :: lib
    integer :: i
    do i = 1, size (lib%entry)
      associate (entry => lib%entry(i))
        if (entry%i_external /= 0) then
          call entry%connect (lib%driver, entry%i_external)
        end if
      end associate
    end do
  end subroutine process_library_load_entries

```

Unload the library, if possible. This reverts the status to “linked”.

```

<Process libraries: process library: TBP>+≡
  procedure :: unload => process_library_unload

<Process libraries: procedures>+≡
  subroutine process_library_unload (lib)
    class(process_library_t), intent(inout) :: lib
    select case (lib%status)
    case (STAT_ACTIVE)
      if (lib%external) then
        call msg_message ("Process library '" // char (lib%basename) &
          // "': unloading")
        call lib%driver%unload ()
      end if
      lib%entry%status = STAT_LINKED
      lib%status = STAT_LINKED
    end select
  end subroutine process_library_unload

```

Unload, clean all generated files and revert the library status. If distclean is set, also remove the makefile and the driver source.

```

<Process libraries: process library: TBP>+≡
  procedure :: clean => process_library_clean

```



```

<Process libraries: procedures>+≡
subroutine process_library_clean (lib, os_data, distclean)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: distclean
  call lib%unload ()
  if (lib%external) then
    call msg_message ("Process library '" // char (lib%basename) &
      // "': removing old files")
    if (distclean) then
      call lib%driver%distclean (os_data)
    else
      call lib%driver%clean (os_data)
    end if
  end if
  where (lib%entry%i_external /= 0)
    lib%entry%status = STAT_CONFIGURED
  elsewhere
    lib%entry%status = STAT_LINKED
  end where
  if (lib%external) then
    lib%status = STAT_CONFIGURED
  else
    lib%status = STAT_LINKED
  end if
end subroutine process_library_clean

```

Unload and revert the library status to INITIAL. This allows for appending new processes. No files are deleted.

```

<Process libraries: process library: TBP>+≡
procedure :: open => process_library_open

<Process libraries: procedures>+≡
subroutine process_library_open (lib)
  class(process_library_t), intent(inout) :: lib
  select case (lib%status)
  case (STAT_OPEN)
  case default
    call lib%unload ()
    lib%entry%status = STAT_OPEN
    lib%status = STAT_OPEN
    call msg_message ("Process library '" // char (lib%basename) &
      // "': open")
  end select
end subroutine process_library_open

```

#### 14.4.4 Use the library

Return the base name of the library

```

<Process libraries: process library: TBP>+≡
procedure :: get_name => process_library_get_name

```

```

<Process libraries: procedures>+≡
function process_library_get_name (lib) result (name)
  class(process_library_t), intent(in) :: lib
  type(string_t) :: name
  name = lib%basename
end function process_library_get_name

```

Once activated, we view the process library object as an interface for accessing the matrix elements.

```

<Process libraries: process library: TBP>+≡
  procedure :: is_active => process_library_is_active

<Process libraries: procedures>+≡
function process_library_is_active (lib) result (flag)
  logical :: flag
  class(process_library_t), intent(in) :: lib
  flag = lib%status == STAT_ACTIVE
end function process_library_is_active

```

Retrieve constants using the process library driver. We assume that the process code has been loaded, if external.

```

<Process libraries: process library entry: TBP>+≡
  procedure :: fill_constants => process_library_entry_fill_constants

<Process libraries: procedures>+≡
subroutine process_library_entry_fill_constants (entry, driver, data)
  class(process_library_entry_t), intent(in) :: entry
  type(prclib_driver_t), intent(in) :: driver
  type(process_constants_t), intent(out) :: data
  integer :: i
  if (entry%i_external /= 0) then
    i = entry%i_external
    data%id = driver%get_process_id (i)
    data%model_name = driver%get_model_name (i)
    data%md5sum = driver%get_md5sum (i)
    data%openmp_supported = driver%get_openmp_status (i)
    data%n_in = driver%get_n_in (i)
    data%n_out = driver%get_n_out (i)
    data%n_flv = driver%get_n_flv (i)
    data%n_hel = driver%get_n_hel (i)
    data%n_col = driver%get_n_col (i)
    data%n_cin = driver%get_n_cin (i)
    data%n_cf = driver%get_n_cf (i)
    call driver%set_flv_state (i, data%flv_state)
    call driver%set_hel_state (i, data%hel_state)
    call driver%set_col_state (i, data%col_state, data%ghost_flag)
    call driver%set_color_factors (i, data%color_factors, data%cf_index)
  else
    select type (proc_driver => entry%driver)
    class is (process_driver_internal_t)
      call proc_driver%fill_constants (data)
    end select
  end if
end subroutine process_library_entry_fill_constants

```

Retrieve the constants and a connected driver for a process, identified by a process ID and a subprocess index. We scan the process entries until we have found a match.

```

<Process libraries: process library: TBP>+≡
    procedure :: connect_process => process_library_connect_process

<Process libraries: procedures>+≡
    subroutine process_library_connect_process &
        (lib, id, i_component, data, proc_driver)
        class(process_library_t), intent(in) :: Lib
        type(string_t), intent(in) :: id
        integer, intent(in) :: i_component
        type(process_constants_t), intent(out) :: data
        class(prc_core_driver_t), allocatable, intent(out) :: proc_driver
        integer :: i
        do i = 1, size (lib%entry)
            associate (entry => lib%entry(i))
                if (entry%def%id == id .and. entry%i_component == i_component) then
                    call entry%fill_constants (lib%driver, data)
                    allocate (proc_driver, source=entry%driver)
                    return
                end if
            end associate
        end do
        ! If we arrive here: Process not found
    end subroutine process_library_connect_process

```

#### 14.4.5 Test

This is the master for calling self-test procedures.

```

<Process libraries: public>+≡
    public :: process_libraries_test

<Process libraries: tests>≡
    subroutine process_libraries_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Process libraries: execute tests>
    end subroutine process_libraries_test

```

#### Empty process list

Test 1: Write an empty process list.

```

<Process libraries: execute tests>≡
    call test (process_libraries_1, "process_libraries_1", &
        "empty process list", &
        u, results)

```

```

<Process libraries: tests>+≡
  subroutine process_libraries_1 (u)
    integer, intent(in) :: u
    type(process_def_list_t) :: process_def_list

    write (u, "(A)")  "* Test output: process_libraries_1"
    write (u, "(A)")  "*   Purpose: Display an empty process definition list"
    write (u, "(A)")

    call process_def_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_1"
  end subroutine process_libraries_1

```

## Process definition list

Test 2: Process definition list with processes and components. Construct the list, write to file, read it in again, and display. Finalize and delete the list after use.

We define a trivial 'test' type for the process variant. The test type contains just one (meaningless) data item, which is an integer.

```

<Process libraries: test declarations>≡
  type, extends (prc_core_def_t) :: prcdef_2_t
    integer :: data = 0
    logical :: file = .false.
  contains
    <Process libraries: prcdef 2: TBP>
  end type prcdef_2_t

```

The process variant is named 'test'.

```

<Process libraries: prcdef 2: TBP>≡
  procedure, nopass :: type_string => prcdef_2_type_string

<Process libraries: tests>+≡
  function prcdef_2_type_string () result (string)
    type(string_t) :: string
    string = "test"
  end function prcdef_2_type_string

```

Write the contents (the integer value).

```

<Process libraries: prcdef 2: TBP>+≡
  procedure :: write => prcdef_2_write

<Process libraries: tests>+≡
  subroutine prcdef_2_write (object, unit)
    class(prcdef_2_t), intent(in) :: object
    integer, intent(in) :: unit
    write (unit, "(3x,A,I0)") "Test data          = ", object%data
  end subroutine prcdef_2_write

```

Recover the integer value.

```
<Process libraries: prcdef 2: TBP>+≡  
  procedure :: read => prcdef_2_read  
  
<Process libraries: tests>+≡  
  subroutine prcdef_2_read (object, unit)  
    class(prcdef_2_t), intent(out) :: object  
    integer, intent(in) :: unit  
    character(80) :: buffer  
    read (unit, "(A)") buffer  
    call strip_prefix (buffer)  
    read (buffer, *) object%data  
  end subroutine prcdef_2_read
```

No external procedures.

```
<Process libraries: prcdef 2: TBP>+≡  
  procedure, nopass :: get_features => prcdef_2_get_features  
  
<Process libraries: tests>+≡  
  subroutine prcdef_2_get_features (features)  
    type(string_t), dimension(:), allocatable, intent(out) :: features  
    allocate (features (0))  
  end subroutine prcdef_2_get_features
```

No code generated.

```
<Process libraries: prcdef 2: TBP>+≡  
  procedure :: generate_code => prcdef_2_generate_code  
  
<Process libraries: tests>+≡  
  subroutine prcdef_2_generate_code (object, &  
    basename, model_name, prt_in, prt_out)  
    class(prcdef_2_t), intent(in) :: object  
    type(string_t), intent(in) :: basename  
    type(string_t), intent(in) :: model_name  
    type(string_t), dimension(:), intent(in) :: prt_in  
    type(string_t), dimension(:), intent(in) :: prt_out  
  end subroutine prcdef_2_generate_code
```

Allocate the driver with the appropriate type.

```
<Process libraries: prcdef 2: TBP>+≡  
  procedure :: allocate_driver => prcdef_2_allocate_driver  
  
<Process libraries: tests>+≡  
  subroutine prcdef_2_allocate_driver (object, driver, basename)  
    class(prcdef_2_t), intent(in) :: object  
    class(prc_core_driver_t), intent(out), allocatable :: driver  
    type(string_t), intent(in) :: basename  
    allocate (prctest_2_t :: driver)  
  end subroutine prcdef_2_allocate_driver
```

Nothing to connect.

```
<Process libraries: prcdef 2: TBP>+≡  
  procedure :: connect => prcdef_2_connect
```

```

<Process libraries: tests>+≡
  subroutine prcdef_2_connect (def, lib_driver, i, proc_driver)
    class(prcdef_2_t), intent(in) :: def
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
  end subroutine prcdef_2_connect

```

The associated driver type.

```

<Process libraries: types>+≡
  type, extends (process_driver_internal_t) :: prctest_2_t
  contains
    <Process libraries: prctest 2: TBP>
  end type prctest_2_t

```

Return the type name.

```

<Process libraries: prctest 2: TBP>≡
  procedure, nopass :: type_name => prctest_2_type_name

<Process libraries: procedures>+≡
  function prctest_2_type_name () result (type)
    type(string_t) :: type
    type = "test"
  end function prctest_2_type_name

```

This should fill constant process data. We do not check those here, however, therefore nothing done.

```

<Process libraries: prctest 2: TBP>+≡
  procedure :: fill_constants => prctest_2_fill_constants

<Process libraries: procedures>+≡
  subroutine prctest_2_fill_constants (driver, data)
    class(prctest_2_t), intent(in) :: driver
    type(process_constants_t), intent(out) :: data
  end subroutine prctest_2_fill_constants

```

Here is the actual test.

For reading, we need a list of templates, i.e., an array containing allocated objects for all available process variants. This is the purpose of `process_core_templates`. Here, we have only a single template for the 'test' variant.

```

<Process libraries: execute tests>+≡
  call test (process_libraries_2, "process_libraries_2", &
    "process definition list", &
    u, results)

<Process libraries: tests>+≡
  subroutine process_libraries_2 (u)
    integer, intent(in) :: u
    type(prc_template_t), dimension(:), allocatable :: process_core_templates
    type(process_def_list_t) :: process_def_list
    type(process_def_entry_t), pointer :: entry => null ()
    class(prc_core_def_t), allocatable :: test_def
    integer :: scratch_unit

```

```

write (u, "(A)")  "*" Test output: process_libraries_2"
write (u, "(A)")  "*" Purpose: Construct a process definition list,"
write (u, "(A)")  "*"           write it to file and reread it"
write (u, "(A)")  ""
write (u, "(A)")  "*" Construct a process definition list"
write (u, "(A)")  "*"   First process definition: empty"
write (u, "(A)")  "*"   Second process definition: two components"
write (u, "(A)")  "*"       First component: empty"
write (u, "(A)")  "*"       Second component: test data"
write (u, "(A)")  "*"   Third process definition:"
write (u, "(A)")  "*"       Embedded decays and polarization"
write (u, "(A)")

allocate (process_core_templates (1))
allocate (prcdef_2_t :: process_core_templates(1)%core_def)

allocate (entry)
call entry%init (var_str ("first"), n_in = 0, n_components = 0)
call entry%compute_md5sum ()
call process_def_list%append (entry)

allocate (entry)
call entry%init (var_str ("second"), model_name = var_str ("Test"), &
    n_in = 1, n_components = 2)
allocate (prcdef_2_t :: test_def)
select type (test_def)
type is (prcdef_2_t); test_def%data = 42
end select
call entry%import_component (2, n_out = 2, &
    prt_in  = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
    method  = var_str ("test"), &
    variant = test_def)
call entry%compute_md5sum ()
call process_def_list%append (entry)

allocate (entry)
call entry%init (var_str ("third"), model_name = var_str ("Test"), &
    n_in = 2, n_components = 1)
allocate (prcdef_2_t :: test_def)
call entry%import_component (1, n_out = 3, &
    prt_in  = &
        new_prt_spec ([var_str ("a"), var_str ("b")]), &
    prt_out = &
        [new_prt_spec (var_str ("c")), &
        new_prt_spec (var_str ("d"), .true.), &
        new_prt_spec (var_str ("e"), [var_str ("e_decay")])], &
    method  = var_str ("test"), &
    variant = test_def)
call entry%compute_md5sum ()
call process_def_list%append (entry)
call process_def_list%write (u)

```

```

write (u, "(A)") ""
write (u, "(A)")  "* Write the process definition list to (scratch) file"

scratch_unit = free_unit ()
open (unit = scratch_unit, status="scratch", action = "readwrite")
call process_def_list%write (scratch_unit)
call process_def_list%final ()

write (u, "(A)")  "* Reread it"
write (u, "(A)")  ""

rewind (scratch_unit)
call process_def_list%read (scratch_unit, process_core_templates)
close (scratch_unit)

call process_def_list%write (u)
call process_def_list%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_2"
end subroutine process_libraries_2

```

### Process library object

Test 3: Process library object with several process definitions and library entries. Just construct the object, modify some initial content, and write the result. The modifications are mostly applied directly, so we do not test anything but the contents and the output routine.

*(Process libraries: execute tests)+≡*

```

call test (process_libraries_3, "process_libraries_3", &
  "recover process definition list from file", &
  u, results)

```

*(Process libraries: tests)+≡*

```

subroutine process_libraries_3 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry

  write (u, "(A)")  "* Test output: process_libraries_3"
  write (u, "(A)")  "* Purpose: Construct a process library object &
    &with entries"
  write (u, "(A)")  ""
  write (u, "(A)")  "* Construct and display a process library object"
  write (u, "(A)")  "*   with 5 entries"
  write (u, "(A)")  "*   associated with 3 matrix element codes"
  write (u, "(A)")  "*   corresponding to 3 process definitions"
  write (u, "(A)")  "*   with 2, 1, 1 components, respectively"
  write (u, "(A)")

  call lib%init (var_str ("testlib"))

  lib%status = STAT_ACTIVE

```



```

lib%n_entries = 5
allocate (lib%entry (lib%n_entries))

allocate (entry)
call entry%init (var_str ("test_a"), n_in = 2, n_components = 2)
call lib%entry(3)%init (STAT_SOURCE, entry%process_def_t, 2, 2)
allocate (prctest_2_t :: lib%entry(3)%driver)
call lib%entry(4)%init (STAT_COMPILED, entry%process_def_t, 1, 0)
call lib%append (entry)

allocate (entry)
call entry%init (var_str ("test_b"), n_in = 2, n_components = 1)
call lib%entry(2)%init (STAT_CONFIGURED, entry%process_def_t, 1, 1)
call lib%append (entry)

allocate (entry)
call entry%init (var_str ("test_c"), n_in = 2, n_components = 1)
call lib%entry(5)%init (STAT_LINKED, entry%process_def_t, 1, 3)
allocate (prctest_2_t :: lib%entry(5)%driver)
call lib%append (entry)

call lib%write (u)
call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_3"
end subroutine process_libraries_3

```

### Process library for test matrix element (no file)

Test 4: We proceed through the library generation and loading phases with a test matrix element type that needs no code written on file.

```

<Process libraries: execute tests>+≡
  call test (process_libraries_4, "process_libraries_4", &
    "build and load internal process library", &
    u, results)

<Process libraries: tests>+≡
  subroutine process_libraries_4 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    type(process_def_entry_t), pointer :: entry
    class(prc_core_def_t), allocatable :: core_def
    type(os_data_t) :: os_data

    write (u, "(A)")  "* Test output: process_libraries_4"
    write (u, "(A)")  "* Purpose: build a process library with an &
      &internal (pseudo) matrix element"
    write (u, "(A)")  "*           No Makefile or code should be generated"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry &
      &(no external code)"

```

```

write (u, "(A)")
call lib%init (var_str ("proclibs4"))

allocate (prcdef_2_t :: core_def)

allocate (entry)
call entry%init (var_str ("proclibs4_a"), n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure ()

write (u, "(A)")  "* Compute MD5 sum"
write (u, "(A)")
call lib%compute_md5sum ()

write (u, "(A)")  "* Write makefile (no-op)"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code (no-op)"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code (no-op)"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(A)")  "* Compile (no-op)"
write (u, "(A)")
call lib%make_compile (os_data)

write (u, "(A)")  "* Link (no-op)"
write (u, "(A)")
call lib%make_link (os_data)

write (u, "(A)")  "* Load (no-op)"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)
call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_4"
end subroutine process_libraries_4

```

## Build workflow for test matrix element

Test 5: We write source code for a dummy process.

We define another trivial type for the process variant. The test type contains just no variable data, but produces code on file.

```
<Process libraries: test declarations>+≡
  type, extends (prc_core_def_t) :: prcdef_5_t
    contains
      <Process libraries: prcdef 5: TBP>
    end type prcdef_5_t
```

The process variant is named `test_file`.

```
<Process libraries: prcdef 5: TBP>≡
  procedure, nopass :: type_string => prcdef_5_type_string

<Process libraries: tests>+≡
  function prcdef_5_type_string () result (string)
    type(string_t) :: string
    string = "test_file"
  end function prcdef_5_type_string
```

We reuse the writer `test_writer_4` from the previous module.

```
<Process libraries: prcdef 5: TBP>+≡
  procedure :: init => prcdef_5_init

<Process libraries: tests>+≡
  subroutine prcdef_5_init (object)
    class(prcdef_5_t), intent(out) :: object
    allocate (test_writer_4_t :: object%writer)
  end subroutine prcdef_5_init
```

Nothing to write.

```
<Process libraries: prcdef 5: TBP>+≡
  procedure :: write => prcdef_5_write

<Process libraries: tests>+≡
  subroutine prcdef_5_write (object, unit)
    class(prcdef_5_t), intent(in) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_5_write
```

Nothing to read.

```
<Process libraries: prcdef 5: TBP>+≡
  procedure :: read => prcdef_5_read

<Process libraries: tests>+≡
  subroutine prcdef_5_read (object, unit)
    class(prcdef_5_t), intent(out) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_5_read
```

Allocate the driver with the appropriate type.

```
<Process libraries: prcdef 5: TBP>+≡
  procedure :: allocate_driver => prcdef_5_allocate_driver
```

```

<Process libraries: tests>+≡
  subroutine prcdef_5_allocate_driver (object, driver, basename)
    class(prcdef_5_t), intent(in) :: object
    class(prc_core_driver_t), intent(out), allocatable :: driver
    type(string_t), intent(in) :: basename
    allocate (prctest_5_t :: driver)
  end subroutine prcdef_5_allocate_driver

```

This time we need code:

```

<Process libraries: prcdef 5: TBP>+≡
  procedure, nopass :: needs_code => prcdef_5_needs_code

<Process libraries: tests>+≡
  function prcdef_5_needs_code () result (flag)
    logical :: flag
    flag = .true.
  end function prcdef_5_needs_code

```

For the test case, we implement a single feature proc1.

```

<Process libraries: prcdef 5: TBP>+≡
  procedure, nopass :: get_features => prcdef_5_get_features

<Process libraries: tests>+≡
  subroutine prcdef_5_get_features (features)
    type(string_t), dimension(:), allocatable, intent(out) :: features
    allocate (features (1))
    features = [ var_str ("proc1") ]
  end subroutine prcdef_5_get_features

```

Nothing to connect.

```

<Process libraries: prcdef 5: TBP>+≡
  procedure :: connect => prcdef_5_connect

<Process libraries: tests>+≡
  subroutine prcdef_5_connect (def, lib_driver, i, proc_driver)
    class(prcdef_5_t), intent(in) :: def
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
  end subroutine prcdef_5_connect

```

The driver type.

```

<Process libraries: types>+≡
  type, extends (prc_core_driver_t) :: prctest_5_t
    contains
    <Process libraries: prctest 5: TBP>
  end type prctest_5_t

```

Return the type name.

```

<Process libraries: prctest 5: TBP>≡
  procedure, nopass :: type_name => prctest_5_type_name

```

```

(Process libraries: tests)+≡
function prctest_5_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_5_type_name

```

Here is the actual test:

```

(Process libraries: execute tests)+≡
call test (process_libraries_5, "process_libraries_5", &
  "build external process library", &
  u, results)

(Process libraries: tests)+≡
subroutine process_libraries_5 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_def_t), allocatable :: core_def
  type(os_data_t) :: os_data

  write (u, "(A)")  "* Test output: process_libraries_5"
  write (u, "(A)")  "* Purpose: build a process library with an &
    &external (pseudo) matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("proclibs5"))
  call os_data_init (os_data)

  allocate (prcdef_5_t :: core_def)
  select type (core_def)
  type is (prcdef_5_t)
    call core_def%init ()
  end select

  allocate (entry)
  call entry%init (var_str ("proclibs5_a"), &
    model_name = var_str ("Test_Model"), &
    n_in = 1, n_components = 1)
  call entry%import_component (1, n_out = 2, &
    prt_in = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
    method = var_str ("test"), &
    variant = core_def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"
  write (u, "(A)")
  call lib%configure ()

  write (u, "(A)")  "* Compute MD5 sum"
  write (u, "(A)")
  call lib%compute_md5sum ()

```

```

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(A)")  "* Compile"
write (u, "(A)")
call lib%make_compile (os_data)

write (u, "(A)")  "* Link"
write (u, "(A)")
call lib%make_link (os_data)

call lib%write (u)

call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_5"
end subroutine process_libraries_5

```

## Build and load library with test matrix element

Test 6: We write source code for a dummy process.

This process variant is identical to the previous case, but it supports a driver for the test procedure 'proc1'.

```

<Process libraries: test declarations>+≡
  type, extends (prc_core_def_t) :: prcdef_6_t
  contains
    <Process libraries: prcdef 6: TBP>
  end type prcdef_6_t

```

The process variant is named `test_file`.

```

<Process libraries: prcdef 6: TBP>≡
  procedure, nopass :: type_string => prcdef_6_type_string

<Process libraries: tests>+≡
  function prcdef_6_type_string () result (string)
    type(string_t) :: string
    string = "test_file"
  end function prcdef_6_type_string

```

We reuse the writer `test_writer_4` from the previous module.

```
<Process libraries: prcdef 6: TBP>+≡
  procedure :: init => prcdef_6_init

<Process libraries: tests>+≡
  subroutine prcdef_6_init (object)
    class(prcdef_6_t), intent(out) :: object
    allocate (test_writer_4_t :: object%writer)
    call object%writer%init_test ()
  end subroutine prcdef_6_init
```

Nothing to write.

```
<Process libraries: prcdef 6: TBP>+≡
  procedure :: write => prcdef_6_write

<Process libraries: tests>+≡
  subroutine prcdef_6_write (object, unit)
    class(prcdef_6_t), intent(in) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_6_write
```

Nothing to read.

```
<Process libraries: prcdef 6: TBP>+≡
  procedure :: read => prcdef_6_read

<Process libraries: tests>+≡
  subroutine prcdef_6_read (object, unit)
    class(prcdef_6_t), intent(out) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_6_read
```

Allocate the driver with the appropriate type.

```
<Process libraries: prcdef 6: TBP>+≡
  procedure :: allocate_driver => prcdef_6_allocate_driver

<Process libraries: tests>+≡
  subroutine prcdef_6_allocate_driver (object, driver, basename)
    class(prcdef_6_t), intent(in) :: object
    class(prc_core_driver_t), intent(out), allocatable :: driver
    type(string_t), intent(in) :: basename
    allocate (prctest_6_t :: driver)
  end subroutine prcdef_6_allocate_driver
```

This time we need code:

```
<Process libraries: prcdef 6: TBP>+≡
  procedure, nopass :: needs_code => prcdef_6_needs_code

<Process libraries: tests>+≡
  function prcdef_6_needs_code () result (flag)
    logical :: flag
    flag = .true.
  end function prcdef_6_needs_code
```

For the test case, we implement a single feature `proc1`.

```

(Process libraries: prcdef 6: TBP)+≡
  procedure, nopass :: get_features => prcdef_6_get_features

(Process libraries: tests)+≡
  subroutine prcdef_6_get_features (features)
    type(string_t), dimension(:), allocatable, intent(out) :: features
    allocate (features (1))
    features = [ var_str ("proc1") ]
  end subroutine prcdef_6_get_features

```

The interface of the only specific feature.

```

(Process libraries: interfaces)≡
  abstract interface
    subroutine proc1_t (n) bind(C)
      import
      integer(c_int), intent(out) :: n
    end subroutine proc1_t
  end interface

```

Connect the feature `proc1` with the process driver.

```

(Process libraries: prcdef 6: TBP)+≡
  procedure :: connect => prcdef_6_connect

(Process libraries: tests)+≡
  subroutine prcdef_6_connect (def, lib_driver, i, proc_driver)
    class(prcdef_6_t), intent(in) :: def
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
    integer(c_int) :: pid, fid
    type(c_funptr) :: fptr
    select type (proc_driver)
    type is (prctest_6_t)
      pid = i
      fid = 1
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%proc1)
    end select
  end subroutine prcdef_6_connect

```

The driver type.

```

(Process libraries: types)+≡
  type, extends (prc_core_driver_t) :: prctest_6_t
    procedure(proc1_t), nopass, pointer :: proc1 => null ()
  contains
    (Process libraries: prctest 6: TBP)
  end type prctest_6_t

```

Return the type name.

```

(Process libraries: prctest 6: TBP)≡
  procedure, nopass :: type_name => prctest_6_type_name

```



```

(Process libraries: tests)+≡
function prctest_6_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_6_type_name

```

Here is the actual test:

```

(Process libraries: execute tests)+≡
call test (process_libraries_6, "process_libraries_6", &
  "build and load external process library", &
  u, results)

(Process libraries: tests)+≡
subroutine process_libraries_6 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_def_t), allocatable :: core_def
  type(os_data_t) :: os_data
  type(string_t), dimension(:), allocatable :: name_list
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: proc_driver
  integer :: i
  integer(c_int) :: n

  write (u, "(A)")  "* Test output: process_libraries_6"
  write (u, "(A)")  "* Purpose: build and load a process library"
  write (u, "(A)")  "*           with an external (pseudo) matrix element"
  write (u, "(A)")  "*           Check single-call linking"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("proclibs6"))
  call os_data_init (os_data)

  allocate (prcdef_6_t :: core_def)
  select type (core_def)
  type is (prcdef_6_t)
    call core_def%init ()
  end select

  allocate (entry)
  call entry%init (var_str ("proclibs6_a"), &
    model_name = var_str ("Test_model"), &
    n_in = 1, n_components = 1)
  call entry%import_component (1, n_out = 2, &
    prt_in = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
    method = var_str ("test"), &
    variant = core_def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"

```

```

write (u, "(A)")
call lib%configure ()

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,A,A)")  "name                = '", &
    char (lib%get_name ()), "'"
write (u, "(1x,A,L1)")  "is active                = ", &
    lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes                = ", &
    lib%get_n_processes ()
write (u, "(1x,A)", advance="no")  "processes                ="
call lib%get_process_id_list (name_list)
do i = 1, size (name_list)
    write (u, "(1x,A)", advance="no")  char (name_list(i))
end do
write (u, *)
write (u, "(1x,A,L1)")  "proclibs6_a is process        = ", &
    lib%contains (var_str ("proclibs6_a"))
write (u, "(1x,A,I0)")  "proclibs6_a has index        = ", &
    lib%get_entry_index (var_str ("proclibs6_a"))
write (u, "(1x,A,L1)")  "foobar is process            = ", &
    lib%contains (var_str ("foobar"))
write (u, "(1x,A,I0)")  "foobar has index            = ", &
    lib%get_entry_index (var_str ("foobar"))
write (u, "(1x,A,I0)")  "n_in(proclibs6_a)            = ", &
    lib%get_n_in (var_str ("proclibs6_a"))
write (u, "(1x,A,A)")  "model_name(proclibs6_a)        = ", &
    char (lib%get_model_name (var_str ("proclibs6_a")))
write (u, "(1x,A,I0)")  "n_components(proclibs6_a) = ", &
    lib%get_n_components (var_str ("proclibs6_a"))
write (u, "(1x,A)", advance="no")  "components(proclibs6_a) ="
call lib%get_component_list (var_str ("proclibs6_a"), name_list)
do i = 1, size (name_list)
    write (u, "(1x,A)", advance="no")  char (name_list(i))
end do
write (u, *)

```

```

write (u, "(A)")
write (u, "(A)")  "* Constants of proclibs6_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("proclibs6_a"), 1, data, proc_driver)

write (u, "(1x,A,A)") "component ID      = ", char (data%id)
write (u, "(1x,A,A)") "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)") "md5sum          = '", data%md5sum, "'"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)") "n_in      = ", data%n_in
write (u, "(1x,A,I0)") "n_out     = ", data%n_out
write (u, "(1x,A,I0)") "n_flv    = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel    = ", data%n_hel
write (u, "(1x,A,I0)") "n_col    = ", data%n_col
write (u, "(1x,A,I0)") "n_cin    = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf     = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I0))") "hel state =", data%hel_state
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")  "* Call feature of proclibs6_a:"
write (u, "(A)")

select type (proc_driver)
type is (prctest_6_t)
    call proc_driver%proc1 (n)
    write (u, "(1x,A,I0)") "proc1 = ", n
end select

call lib%final ( )

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_6"
end subroutine process_libraries_6

```

## MD5 sums

Check MD5 sum calculation.

*<Process libraries: execute tests>+≡*

```

call test (process_libraries_7, "process_libraries_7", &
    "process definition list", &
    u, results)

```

*<Process libraries: tests>+≡*

```

subroutine process_libraries_7 (u)
    integer, intent(in) :: u
    type(prc_template_t), dimension(:), allocatable :: process_core_templates
    type(process_def_entry_t) :: entry

```

```

class(prc_core_def_t), allocatable :: test_def

write (u, "(A)")  "* Test output: process_libraries_7"
write (u, "(A)")  "* Purpose: Construct a process definition list &
                    &and check MD5 sums"
write (u, "(A)")
write (u, "(A)")  "* Construct a process definition list"
write (u, "(A)")  "*   Process: two components"
write (u, "(A)")

allocate (process_core_templates (1))
allocate (prcdef_2_t :: process_core_templates(1)%core_def)

call entry%init (var_str ("first"), model_name = var_str ("Test"), &
                n_in = 1, n_components = 2)
allocate (prcdef_2_t :: test_def)
select type (test_def)
type is (prcdef_2_t); test_def%data = 31
end select
call entry%import_component (1, n_out = 3, &
    prt_in  = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c"), &
                            var_str ("e")]), &
    method  = var_str ("test"), &
    variant = test_def)
allocate (prcdef_2_t :: test_def)
select type (test_def)
type is (prcdef_2_t); test_def%data = 42
end select
call entry%import_component (2, n_out = 2, &
    prt_in  = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
    method  = var_str ("test"), &
    variant = test_def)
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute MD5 sums"
write (u, "(A)")

call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recalculate MD5 sums (should be identical)"
write (u, "(A)")

call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Modify a component and recalculate MD5 sums"
write (u, "(A)")

```

```

select type (test_def => entry%initial(2)%core_def)
type is (prcdef_2_t)
    test_def%data = 54
end select
call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Modify the model and recalculate MD5 sums"
write (u, "(A)")

entry%model_name = "foo"
call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_7"
end subroutine process_libraries_7

```

Here is the actual test:

```

<Process libraries: execute tests>+≡
    call test (process_libraries_8, "process_libraries_8", &
        "library status checks", &
        u, results)

<Process libraries: tests>+≡
subroutine process_libraries_8 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    type(process_def_entry_t), pointer :: entry
    class(prc_core_def_t), allocatable :: core_def
    type(os_data_t) :: os_data
    type(string_t), dimension(:), allocatable :: name_list
    type(process_constants_t) :: data
    class(prc_core_driver_t), allocatable :: proc_driver
    integer :: i
    integer(c_int) :: n

    write (u, "(A)")  "* Test output: process_libraries_8"
    write (u, "(A)")  "* Purpose: build and load a process library"
    write (u, "(A)")  "*           with an external (pseudo) matrix element"
    write (u, "(A)")  "*           Check status updates"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call lib%init (var_str ("proclibs8"))
    call os_data_init (os_data)

    allocate (prcdef_6_t :: core_def)
    select type (core_def)
    type is (prcdef_6_t)
        call core_def%init ()
    end select

```

```

allocate (entry)
call entry%init (var_str ("proclibs8_a"), &
    model_name = var_str ("Test_model"), &
    n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
    prt_in = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
    method = var_str ("test"), &
    variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")

call lib%configure ()
call lib%compute_md5sum ()

associate (def => lib%entry(1)%def%initial(1))
    select type (writer => lib%driver%record(1)%writer)
        type is (test_writer_4_t)
            writer%md5sum = def%md5sum
        end select
end associate

write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .false.)

write (u, "(A)")  "* Write process source code"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status

```

```

write (u, "(A)")
write (u, "(A)")  "* Compile and load"
write (u, "(A)")

call lib%load (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Append process and reconfigure"
write (u, "(A)")

allocate (prcdef_6_t :: core_def)
select type (core_def)
type is (prcdef_6_t)
    call core_def%init ()
end select

allocate (entry)
call entry%init (var_str ("proclibs8_b"), &
    model_name = var_str ("Test_model"), &
    n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
    prt_in = new_prt_spec ([var_str ("a")]), &
    prt_out = new_prt_spec ([var_str ("b"), var_str ("d")]), &
    method = var_str ("test"), &
    variant = core_def)
call lib%append (entry)

call lib%configure ()
call lib%compute_md5sum ()
associate (def => lib%entry(2)%def%initial(1))
    select type (writer => lib%driver%record(2)%writer)
    type is (test_writer_4_t)
        writer%md5sum = def%md5sum
    end select
end associate
call lib%write_makefile (os_data, force = .false.)
call lib%write_driver (force = .false.)

write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status   = ", lib%status

```

```

write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Write source code"
write (u, "(A)")

call lib%make_source (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Reset status"
write (u, "(A)")

lib%status = STAT_CONFIGURED
lib%entry%status = STAT_CONFIGURED
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Partial cleanup"
write (u, "(A)")

call lib%clean (os_data, distclean = .false.)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)") "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)") "lib status = ", lib%status
write (u, "(1x,A,I0)") "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)") "proc2 status = ", lib%entry(2)%status

```



```

write (u, "(A)")
write (u, "(A)")  "* Complete cleanup"

call lib%clean (os_data, distclean = .true.)
call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_libraries_8"
end subroutine process_libraries_8

```

## 14.5 Process Library Stacks

For storing and handling multiple libraries, we define process library stacks. These are ordinary stacks where new entries are pushed onto the top.

```

<prclib_stacks.f90>≡
  <File header>

  module prclib_stacks

    <Use strings>
    <Use file utils>
    use unit_tests
    use os_interface
    use models

    use process_libraries

    <Standard module head>

    <Prclib stacks: public>

    <Prclib stacks: types>

    contains

    <Prclib stacks: procedures>

    <Prclib stacks: tests>

    end module prclib_stacks

```

### 14.5.1 The stack entry type

A stack entry is a process library object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, process libraries should be prepared as process entry objects.

```

<Prclib stacks: public>≡
  public :: prclib_entry_t

```

```

<Prclib stacks: types>≡
  type, extends (process_library_t) :: prclib_entry_t
    type(prclib_entry_t), pointer :: next => null ()
  end type prclib_entry_t

```

## 14.5.2 The prclib stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

```

<Prclib stacks: public>+≡
  public :: prclib_stack_t

<Prclib stacks: types>+≡
  type :: prclib_stack_t
    integer :: n = 0
    type(prclib_entry_t), pointer :: first => null ()
  contains
    <Prclib stacks: prclib stack: TBP>
  end type prclib_stack_t

```

Finalizer. Iteratively deallocate the stack entries. The resulting empty stack can be immediately recycled, if necessary.

```

<Prclib stacks: prclib stack: TBP>≡
  procedure :: final => prclib_stack_final

<Prclib stacks: procedures>≡
  subroutine prclib_stack_final (object)
    class(prclib_stack_t), intent(inout) :: object
    type(prclib_entry_t), pointer :: lib
    do while (associated (object%first))
      lib => object%first
      object%first => lib%next
      call lib%final ()
      deallocate (lib)
    end do
    object%n = 0
  end subroutine prclib_stack_final

```

Output. The entries on the stack will be ordered LIFO, i.e., backwards.

```

<Prclib stacks: prclib stack: TBP>+≡
  procedure :: write => prclib_stack_write

<Prclib stacks: procedures>+≡
  subroutine prclib_stack_write (object, unit)
    class(prclib_stack_t), intent(in) :: object
    integer, intent(in), optional :: unit
    type(prclib_entry_t), pointer :: lib
    integer :: u
    u = output_unit (unit)
    call write_separator_double (u)
    select case (object%n)
    case (0)

```

```

        write (u, "(1x,A)") "Process library stack: [empty]"
    case default
        write (u, "(1x,A)") "Process library stack:"
        lib => object%first
        do while (associated (lib))
            call write_separator (u)
            call lib%write (u)
            lib => lib%next
        end do
    end select
    call write_separator_double (u)
end subroutine prclib_stack_write

```

### 14.5.3 Operating on Stacks

We take a library entry pointer and push it onto the stack. The previous pointer is nullified. Subsequently, the library entry is ‘owned’ by the stack and will be finalized when the stack is deleted.

```

<Prclib stacks: prclib stack: TBP>+≡
    procedure :: push => prclib_stack_push

<Prclib stacks: procedures>+≡
    subroutine prclib_stack_push (stack, lib)
        class(prclib_stack_t), intent(inout) :: stack
        type(prclib_entry_t), intent(inout), pointer :: lib
        lib%next => stack%first
        stack%first => lib
        lib => null ()
        stack%n = stack%n + 1
    end subroutine prclib_stack_push

```

### 14.5.4 Accessing Contents

Return a pointer to the topmost stack element. The result type is just the bare `process_library_t`. There is no `target` attribute required since the stack elements are allocated via pointers.

```

<Prclib stacks: prclib stack: TBP>+≡
    procedure :: get_first_ptr => prclib_stack_get_first_ptr

<Prclib stacks: procedures>+≡
    function prclib_stack_get_first_ptr (stack) result (ptr)
        class(prclib_stack_t), intent(in) :: stack
        type(process_library_t), pointer :: ptr
        ptr => stack%first%process_library_t
    end function prclib_stack_get_first_ptr

```

Return a complete list of the libraries (names) in the stack. The list is in the order in which the elements got pushed onto the stack, so the ‘first’ entry is listed last.

```

<Prclib stacks: prclib stack: TBP>+≡
    procedure :: get_names => prclib_stack_get_names

```

```

<Prclib stacks: procedures>+≡
subroutine prclib_stack_get_names (stack, libname)
  class(prclib_stack_t), intent(in) :: stack
  type(string_t), dimension(:), allocatable, intent(out) :: libname
  type(prclib_entry_t), pointer :: lib
  integer :: i
  allocate (libname (stack%n))
  i = stack%n
  lib => stack%first
  do while (associated (lib))
    libname(i) = lib%get_name ()
    i = i - 1
    lib => lib%next
  end do
end subroutine prclib_stack_get_names

```

Return a pointer to the library with given name.

```

<Prclib stacks: prclib_stack: TBP>+≡
  procedure :: get_library_ptr => prclib_stack_get_library_ptr

<Prclib stacks: procedures>+≡
function prclib_stack_get_library_ptr (stack, libname) result (ptr)
  class(prclib_stack_t), intent(in) :: stack
  type(string_t), intent(in) :: libname
  type(process_library_t), pointer :: ptr
  type(prclib_entry_t), pointer :: current
  current => stack%first
  do while (associated (current))
    if (current%get_name () == libname) then
      ptr => current%process_library_t
      return
    end if
    current => current%next
  end do
  ptr => null ()
end function prclib_stack_get_library_ptr

```

### 14.5.5 Auxiliary stuff

Write a separator line.

```

<Prclib stacks: procedures>+≡
subroutine write_separator (u)
  integer, intent(in) :: u
  write (u, "(A)") repeat ("-", 72)
end subroutine write_separator

subroutine write_separator_double (u)
  integer, intent(in) :: u
  write (u, "(A)") repeat ("=", 72)
end subroutine write_separator_double

```

### 14.5.6 Unit tests

```
<Prclib stacks: public>+≡
    public :: prclib_stacks_test
<Prclib stacks: tests>≡
    subroutine prclib_stacks_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
<Prclib stacks: execute tests>
    end subroutine prclib_stacks_test
```

#### Write an empty process library stack

The most trivial test is to write an uninitialized process library stack.

```
<Prclib stacks: execute tests>≡
    call test (prclib_stacks_1, "prclib_stacks_1", &
        "write an empty process library stack", &
        u, results)
<Prclib stacks: tests>+≡
    subroutine prclib_stacks_1 (u)
        integer, intent(in) :: u
        type(prclib_stack_t) :: stack

        write (u, "(A)")  "* Test output: prclib_stacks_1"
        write (u, "(A)")  "* Purpose: display an empty process library stack"
        write (u, "(A)")

        call stack%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: prclib_stacks_1"

    end subroutine prclib_stacks_1
```

#### Fill a process library stack

Fill a process library stack with two (identical) processes.

```
<Prclib stacks: execute tests>+≡
    call test (prclib_stacks_2, "prclib_stacks_2", &
        "fill a process library stack", &
        u, results)
<Prclib stacks: tests>+≡
    subroutine prclib_stacks_2 (u)
        integer, intent(in) :: u
        type(prclib_stack_t) :: stack
        type(prclib_entry_t), pointer :: lib

        write (u, "(A)")  "* Test output: prclib_stacks_2"
        write (u, "(A)")  "* Purpose: fill a process library stack"
        write (u, "(A)")
```

```

write (u, "(A)")  "* Initialize two (empty) libraries &
                  &and push them on the stack"
write (u, "(A)")

allocate (lib)
call lib%init (var_str ("lib1"))
call stack%push (lib)

allocate (lib)
call lib%init (var_str ("lib2"))
call stack%push (lib)

call stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call stack%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prclib_stacks_2"

end subroutine prclib_stacks_2

```

## 14.6 Trivial matrix element for tests

For the purpose of testing the workflow, we implement here a matrix element with the simplest possible structure.

This matrix element generator can only generate a single process: a quartic interaction of a massless, neutral and colorless scalar  $\mathbf{s}$  with unit coupling results in a trivial  $2 \rightarrow 2$  scattering process. The matrix element is implemented internally, so we do not need the machinery of external process libraries.

```

⟨prc_test.f90⟩≡
  ⟨File header⟩

module prc_test

  use iso_c_binding !NODEP!
  use kinds !NODEP!
  ⟨Use strings⟩
  use unit_tests
  use os_interface

  use process_constants
  use prclib_interfaces
  use prc_core_def
  use particle_specifiers
  use process_libraries

  ⟨Standard module head⟩

```

```

    <Test ME: public>

    <Test ME: types>

    contains

    <Test ME: procedures>

    <Test ME: tests>

end module prc_test

```

### 14.6.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

```

<Test ME: public>≡
    public :: prc_test_def_t

<Test ME: types>≡
    type, extends (prc_core_def_t) :: prc_test_def_t
        type(string_t) :: model_name
        type(string_t), dimension(:), allocatable :: prt_in
        type(string_t), dimension(:), allocatable :: prt_out
    contains
        <Test ME: test me def: TBP>
    end type prc_test_def_t

<Test ME: test me def: TBP>≡
    procedure, nopass :: type_string => prc_test_def_type_string

<Test ME: procedures>≡
    function prc_test_def_type_string () result (string)
        type(string_t) :: string
        string = "test_me"
    end function prc_test_def_type_string

```

There is no 'feature' here since there is no external code.

```

<Test ME: test me def: TBP>+≡
    procedure, nopass :: get_features => prc_test_def_get_features

<Test ME: procedures>+≡
    subroutine prc_test_def_get_features (features)
        type(string_t), dimension(:), allocatable, intent(out) :: features
        allocate (features (0))
    end subroutine prc_test_def_get_features

```

Initialization: set some data (not really useful).

```

<Test ME: test me def: TBP>+≡
    procedure :: init => prc_test_def_init

```

```

<Test ME: procedures>+≡
subroutine prc_test_def_init (object, model_name, prt_in, prt_out)
  class(prc_test_def_t), intent(out) :: object
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  object%model_name = model_name
  allocate (object%prt_in (size (prt_in)))
  object%prt_in = prt_in
  allocate (object%prt_out (size (prt_out)))
  object%prt_out = prt_out
end subroutine prc_test_def_init

```

Write/read process- and method-specific data. (No-op)

```

<Test ME: test me def: TBP>+≡
  procedure :: write => prc_test_def_write

<Test ME: procedures>+≡
subroutine prc_test_def_write (object, unit)
  class(prc_test_def_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_write

```

```

<Test ME: test me def: TBP>+≡
  procedure :: read => prc_test_def_read

<Test ME: procedures>+≡
subroutine prc_test_def_read (object, unit)
  class(prc_test_def_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_read

```

Allocate the driver for test ME matrix elements. We get the actual component ID (basename), and we can transfer all process-specific data from the process definition.

```

<Test ME: test me def: TBP>+≡
  procedure :: allocate_driver => prc_test_def_allocate_driver

<Test ME: procedures>+≡
subroutine prc_test_def_allocate_driver (object, driver, basename)
  class(prc_test_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prc_test_t :: driver)
  select type (driver)
  type is (prc_test_t)
    driver%id = basename
    driver%model_name = object%model_name
  end select
end subroutine prc_test_def_allocate_driver

```

Nothing to connect. This subroutine will not be called.

```

<Test ME: test me def: TBP>+≡
  procedure :: connect => prc_test_def_connect

```



```

<Test ME: procedures>+≡
  subroutine prc_test_def_connect (def, lib_driver, i, proc_driver)
    class(prc_test_def_t), intent(in) :: def
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
  end subroutine prc_test_def_connect

```

## 14.6.2 Driver

```

<Test ME: public>+≡
  public :: prc_test_t

<Test ME: types>+≡
  type, extends (process_driver_internal_t) :: prc_test_t
    type(string_t) :: id
    type(string_t) :: model_name
  contains
    <Test ME: test me driver: TBP>
  end type prc_test_t

```

In contrast to generic matrix-element implementations, we can hard-wire the amplitude method as a type-bound procedure.

```

<Test ME: test me driver: TBP>≡
  procedure, nopass :: get_amplitude => prc_test_get_amplitude

<Test ME: procedures>+≡
  function prc_test_get_amplitude (p) result (amp)
    complex(default) :: amp
    real(default), dimension(:,:), intent(in) :: p
    amp = 1
  end function prc_test_get_amplitude

```

The reported type is the same as for the `prc_test_def_t` type.

```

<Test ME: test me driver: TBP>+≡
  procedure, nopass :: type_name => prc_test_type_name

<Test ME: procedures>+≡
  function prc_test_type_name () result (string)
    type(string_t) :: string
    string = "test_me"
  end function prc_test_type_name

```

Fill process constants.

```

<Test ME: test me driver: TBP>+≡
  procedure :: fill_constants => prc_test_fill_constants

<Test ME: procedures>+≡
  subroutine prc_test_fill_constants (driver, data)
    class(prc_test_t), intent(in) :: driver
    type(process_constants_t), intent(out) :: data
    data%id = driver%id
  end subroutine prc_test_fill_constants

```

```

data%model_name = driver%model_name
data%n_in = 2
data%n_out = 2
data%n_flv = 1
data%n_hel = 1
data%n_col = 1
data%n_cin = 2
data%n_cf = 1
allocate (data%flv_state (4, 1))
data%flv_state = 25
allocate (data%hel_state (4, 1))
data%hel_state = 0
allocate (data%col_state (2, 4, 1))
data%col_state = 0
allocate (data%ghost_flag (4, 1))
data%ghost_flag = .false.
allocate (data%color_factors (1))
data%color_factors = 1
allocate (data%cf_index (2, 1))
data%cf_index = 1
end subroutine prc_test_fill_constants

```

### 14.6.3 Shortcut

Since this module is there for testing purposes, we set up a subroutine that does all the work at once: create a library with a single process, configure and load, and set up the driver.

```

<Test ME: public>+≡
public :: prc_test_create_library

<Test ME: procedures>+≡
subroutine prc_test_create_library (procname, lib)
  type(string_t), intent(in) :: procname
  type(process_library_t), intent(out) :: lib
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  call lib%init (procname)
  model_name = "Test"
  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("s"), var_str ("s")]
  prt_out = [var_str ("s"), var_str ("s")]
  allocate (prc_test_def_t :: def)
  select type (def)
  type is (prc_test_def_t)
    call def%init (model_name, prt_in, prt_out)
  end select
  allocate (entry)
  call entry%init (procname, model_name = model_name, &
    n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &

```

```

        prt_in = new_prt_spec (prt_in), &
        prt_out = new_prt_spec (prt_out), &
        method = var_str ("test_me"), &
        variant = def)
    call lib%append (entry)
    call lib%configure ()
    call lib%load (os_data)
end subroutine prc_test_create_library

```

#### 14.6.4 Test

This is the master for calling self-test procedures.

```

<Test ME: public>+≡
    public :: prc_test_test

<Test ME: tests>≡
    subroutine prc_test_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Test ME: execute tests>
end subroutine prc_test_test

```

#### Generate and load the process

The process is  $ss \rightarrow ss$ , where  $s$  is a trivial scalar particle, for vanishing mass and unit coupling. We initialize the process, build the library, and compute the particular matrix element for momenta of unit energy and right-angle scattering. (The scattering is independent of angle.) The matrix element is equal to unity.

```

<Test ME: execute tests>≡
    call test (prc_test_1, "prc_test_1", &
        "build and load trivial process", &
        u, results)

<Test ME: tests>+≡
    subroutine prc_test_1 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(process_library_t) :: lib
        class(prc_core_def_t), allocatable :: def
        type(process_def_entry_t), pointer :: entry
        type(string_t) :: model_name
        type(string_t), dimension(:), allocatable :: prt_in, prt_out
        type(process_constants_t) :: data
        class(prc_core_driver_t), allocatable :: driver
        real(default), dimension(0) :: par
        real(default), dimension(0:3,4) :: p
        integer :: i

        write (u, "(A)")  "* Test output: prc_test_1"
        write (u, "(A)")  "*   Purpose: create a trivial process"
        write (u, "(A)")  "*               build a library and &

```

```

        &access the matrix element"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry"
write (u, "(A)")
call lib%init (var_str ("prc_test1"))

model_name = "Test"
allocate (prt_in (2), prt_out (2))
prt_in  = [var_str ("s"), var_str ("s")]
prt_out = [var_str ("s"), var_str ("s")]

allocate (prc_test_def_t :: def)
select type (def)
type is (prc_test_def_t)
    call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("prc_test1_a"), model_name = model_name, &
    n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
    prt_in  = new_prt_spec (prt_in), &
    prt_out = new_prt_spec (prt_out), &
    method  = var_str ("test_me"), &
    variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure ()

write (u, "(A)")  "* Load library"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active"           = ", &
    lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes"         = ", &
    lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of prc_test1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("prc_test1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID"         = ", char (data%id)
write (u, "(1x,A,A)")  "model name"           = ", char (data%model_name)

```

```

write (u, "(1x,A,A,A)") "md5sum          = ', data%md5sum, '"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

!   write (u, "(A)")
!   write (u, "(A)")  "* Initialize driver:"
!   write (u, "(A)")

!   call driver%init (par)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
    1.0_default, 0.0_default, 0.0_default, 1.0_default, &
    1.0_default, 0.0_default, 0.0_default,-1.0_default, &
    1.0_default, 1.0_default, 0.0_default, 0.0_default, &
    1.0_default,-1.0_default, 0.0_default, 0.0_default &
    ], [4,4])
do i = 1, 4
    write (u, "(2x,A,I0,A,4(1x,F7.4))") "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver)
type is (prc_test_t)
    write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
end select

call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_test_1"

end subroutine prc_test_1

```

## Shortcut

This is identical to the previous test, but we create the library be a single command. This is handy for other modules which use the test process.

```
<Test ME: execute tests>+≡
    call test (prc_test_2, "prc_test_2", &
               "build and load trivial process using shortcut", &
               u, results)

<Test ME: tests>+≡
    subroutine prc_test_2 (u)
        integer, intent(in) :: u
        type(process_library_t) :: lib
        class(prc_core_driver_t), allocatable :: driver
        type(process_constants_t) :: data
        real(default), dimension(0:3,4) :: p

        write (u, "(A)")  "* Test output: prc_test_2"
        write (u, "(A)")  "*   Purpose: create a trivial process"
        write (u, "(A)")  "*               build a library and &
                           &access the matrix element"
        write (u, "(A)")

        write (u, "(A)")  "* Build and load a process library with one entry"

        call prc_test_create_library (var_str ("prc_test2"), lib)
        call lib%connect_process (var_str ("prc_test2"), 1, data, driver)

        p = reshape ([ &
                      1.0_default, 0.0_default, 0.0_default, 1.0_default, &
                      1.0_default, 0.0_default, 0.0_default,-1.0_default, &
                      1.0_default, 1.0_default, 0.0_default, 0.0_default, &
                      1.0_default,-1.0_default, 0.0_default, 0.0_default &
                      ], [4,4])

        write (u, "(A)")
        write (u, "(A)")  "* Compute matrix element:"
        write (u, "(A)")

        select type (driver)
        type is (prc_test_t)
            write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
        end select

        call lib%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: prc_test_2"

    end subroutine prc_test_2
```

## Chapter 15

# Integration and Event Generation

This is the central part of the WHIZARD package. It provides the functionality for evaluating structure functions, kinematics and matrix elements, integration and event generation. It combines the various parts that deal with those tasks individually and organizes the data transfer between them.

**integration\_results** This module handles results from integrating processes. It records passes and iterations, calculates statistical averages, and provides the user output of integration results.

**prc\_core** Here, we define the abstract `prc_core_t` type which handles all specific features of kinematics matrix-element evaluation that depend on a particular class of processes.

**parton\_states** A `parton_state_t` object represents an elementary partonic interaction. There are two versions: one for the isolated elementary process, one for the elementary process convoluted with the structure-function chain. The parton state is an effective state. It needs not coincide with the seed-kinematics state which is used in evaluating phase space.

**processes** Here, all pieces are combined for the purpose of evaluating the elementary processes. The whole algorithm is coded in terms of abstract data types as defined in the appropriate modules: `prc_core` for matrix-element evaluation, `prc_core_def` for the associated configuration and driver, `sf_base` for beams and structure-functions, `phs_base` for phase space, and `mci_base` for integration and event generation.

**decays** (not re-activated yet) Organize elementary processes in decay chains for on-shell particles.

**showers** (not re-activated yet) Create QED/QCD showers out of the partons that are emitted by elementary processes. This should be interleaved with showering of radiated particles (structure functions) and multiple interactions.

**hadrons** (not implemented yet) Apply hadronization to the partonic events, interleaved with hadron decays. (The current setup relies on hadronizing partonic events externally.)

**events** (not re-implemented yet) Combine all pieces to generate full events.

## 15.1 Integration results

We record integration results and errors in a dedicated type. This allows us to do further statistics such as weighted average, chi-squared, grouping by integration passes, etc.

Note WHIZARD 2.2.0: This code is taken from the previous `processes` module essentially unchanged and converted into a separate module. It lacks an overhaul and, in particular, self-tests.

```
<integration_results.f90>≡
  module integration_results

    use limits, only: GML_MIN_RANGE_RATIO !NODEP!
    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use md5
    use cputime
    use os_interface
    use mci_base

    <Standard module head>

    <Integration results: public>

    <Integration results: parameters>

    <Integration results: types>

    <Integration results: interfaces>

    contains

    <Integration results: procedures>

  end module integration_results
```

### 15.1.1 Integration results entry

This object collects the results of an integration pass and makes them available to the outside.

The results object has to distinguish the process type:

We store the process type, the index of the integration pass and the absolute iteration index, the number of iterations contained in this result (for averages), and the integral (cross section or partial width), error estimate, efficiency and time estimate.



For intermediate results, we set a flag if this result is an improvement w.r.t. previous ones.

The process type indicates decay or scattering. Dummy entries (skipped iterations) have a process type of PRC\_UNKNOWN.

```

<Integration results: types>≡
  type :: integration_entry_t
    private
    integer :: process_type = PRC_UNKNOWN
    integer :: pass = 0
    integer :: it = 0
    integer :: n_it = 0
    integer :: n_calls = 0
    logical :: improved = .false.
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: efficiency = 0
    real(default) :: chi2 = 0
    real(default), dimension(:), allocatable :: grove_weight
    type(time_t) :: time_start
    type(time_t) :: time_end
  end type integration_entry_t

```

The possible values of the type indicator:

```

<Integration results: parameters>≡
  integer, parameter, public :: PRC_UNKNOWN = 0
  integer, parameter, public :: PRC_DECAY = 1
  integer, parameter, public :: PRC_SCATTERING = 2

```

Initialize with all relevant data

```

<Integration results: procedures>≡
  subroutine integration_entry_init (entry, &
    process_type, pass, it, n_it, n_calls, improved, &
    integral, error, efficiency, chi2, grove_weight, &
    time_start, time_end)
    type(integration_entry_t), intent(out) :: entry
    integer, intent(in) :: process_type, pass, it, n_it, n_calls
    logical, intent(in) :: improved
    real(default), intent(in) :: integral, error, efficiency
    real(default), intent(in), optional :: chi2
    real(default), dimension(:), intent(in), optional :: grove_weight
    type(time_t), intent(in), optional :: time_start, time_end
    integer :: n_groves
    entry%process_type = process_type
    entry%pass = pass
    entry%it = it
    entry%n_it = n_it
    entry%n_calls = n_calls
    entry%improved = improved
    entry%integral = integral
    entry%error = error
    entry%efficiency = efficiency
    if (present (chi2)) &

```

```

        entry%chi2 = chi2
    if (present (grove_weight)) then
        n_groves = size (grove_weight)
        allocate (entry%grove_weight (n_groves))
        entry%grove_weight = grove_weight
    end if
    if (present (time_start) .and. present (time_end)) then
        entry%time_start = time_start
        entry%time_end = time_end
    end if
end subroutine integration_entry_init

```

Access values, some of them computed on demand:

(*Integration results: procedures*) +=

```

    elemental function integration_entry_get_pass (entry) result (n)
        integer :: n
        type(integration_entry_t), intent(in) :: entry
        n = entry%pass
    end function integration_entry_get_pass

    elemental function integration_entry_get_n_calls (entry) result (n)
        integer :: n
        type(integration_entry_t), intent(in) :: entry
        n = entry%n_calls
    end function integration_entry_get_n_calls

    elemental function integration_entry_get_integral (entry) result (int)
        real(default) :: int
        type(integration_entry_t), intent(in) :: entry
        int = entry%integral
    end function integration_entry_get_integral

    elemental function integration_entry_get_error (entry) result (err)
        real(default) :: err
        type(integration_entry_t), intent(in) :: entry
        err = entry%error
    end function integration_entry_get_error

    elemental function integration_entry_get_relative_error (entry) result (err)
        real(default) :: err
        type(integration_entry_t), intent(in) :: entry
        if (entry%integral /= 0) then
            err = entry%error / entry%integral
        else
            err = 0
        end if
    end function integration_entry_get_relative_error

    elemental function integration_entry_get_accuracy (entry) result (acc)
        real(default) :: acc
        type(integration_entry_t), intent(in) :: entry
        acc = accuracy (entry%integral, entry%error, entry%n_calls)
    end function integration_entry_get_accuracy

```

```

elemental function accuracy (integral, error, n_calls) result (acc)
  real(default) :: acc
  real(default), intent(in) :: integral, error
  integer, intent(in) :: n_calls
  if (integral /= 0) then
    acc = error / integral * sqrt (real (n_calls, default))
  else
    acc = 0
  end if
end function accuracy

elemental function integration_entry_get_efficiency (entry) result (eff)
  real(default) :: eff
  type(integration_entry_t), intent(in) :: entry
  eff = entry%efficiency
end function integration_entry_get_efficiency

elemental function integration_entry_get_chi2 (entry) result (chi2)
  real(default) :: chi2
  type(integration_entry_t), intent(in) :: entry
  chi2 = entry%chi2
end function integration_entry_get_chi2

elemental function integration_entry_get_time_per_event (entry) result (tpe)
  real(default) :: tpe
  type(integration_entry_t), intent(in) :: entry
  real(default) :: time_in_seconds
  if (entry%n_calls /= 0 .and. entry%efficiency /= 0) then
    time_in_seconds = entry%time_end - entry%time_start
    tpe = time_in_seconds / entry%n_calls / entry%efficiency
  else
    tpe = 0
  end if
end function integration_entry_get_time_per_event

elemental function integration_entry_has_improved (entry) result (flag)
  logical :: flag
  type(integration_entry_t), intent(in) :: entry
  flag = entry%improved
end function integration_entry_has_improved

elemental function integration_entry_get_n_groves (entry) result (n_groves)
  integer :: n_groves
  type(integration_entry_t), intent(in) :: entry
  if (allocated (entry%grove_weight)) then
    n_groves = size (entry%grove_weight)
  else
    n_groves = 0
  end if
end function integration_entry_get_n_groves

```

Output. This writes the header line for the result account below:

```

<Integration results: procedures>+=
  subroutine write_header (process_type, unit, logfile)

```

```

integer, intent(in) :: process_type
integer, intent(in), optional :: unit
logical, intent(in), optional :: logfile
character(5) :: phys_unit
integer :: u
u = output_unit (unit); if (u < 0) return
select case (process_type)
case (PRC_DECAY);      phys_unit = "[GeV]"
case (PRC_SCATTERING); phys_unit = "[fb] "
case default
    phys_unit = ""
end select
write (msg_buffer, "(A)") &
    "It      Calls Integral" // phys_unit // &
    " Error" // phys_unit // &
    " Err[%]  Acc Eff[%]  Chi2 N[It] |"
call msg_message (unit=u, logfile=logfile)
end subroutine write_header

```

This writes a separator for result display:

*(Integration results: procedures)+≡*

```

subroutine write_hline (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, "(A)")  "|" // (repeat("-", 77)) // "|"
    flush (u)
end subroutine write_hline

subroutine write_dline (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, "(A)")  "|" // (repeat("=", 77)) // "|"
    flush (u)
end subroutine write_dline

```

This writes the standard result account into one screen line. The verbose version uses multiple lines and prints the unabridged values. Dummy entries are not written.

*(Integration results: procedures)+≡*

```

subroutine integration_entry_write (entry, unit, verbose)
    type(integration_entry_t), intent(in) :: entry
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    character(1) :: star
    logical :: verb
    u = output_unit (unit); if (u < 0) return
    verb = .false.; if (present (verbose)) verb = verbose
    if (verb) then
        write (u, *) "process_type = ", entry%process_type
        write (u, *) "                pass = ", entry%pass
    end if
end subroutine integration_entry_write

```

```

write (u, *) "          it = ", entry%it
write (u, *) "          n_it = ", entry%n_it
write (u, *) "          n_calls = ", entry%n_calls
write (u, *) "          improved = ", entry%improved
write (u, *) "          integral = ", entry%integral
write (u, *) "          error = ", entry%error
write (u, *) "          efficiency = ", entry%efficiency
write (u, *) "          chi2 = ", entry%chi2
if (allocated (entry%grove_weight)) then
  write (u, *) "          n_groves = ", size (entry%grove_weight)
  write (u, *) "          grove_weight = ", entry%grove_weight
else
  write (u, *) "          n_groves = 0"
end if
else if (entry%process_type /= PRC_UNKNOWN) then
  if (entry%improved) then
    star = "*"
  else
    star = " "
  end if
1  format (1x, I3, 1x, I10, 1x, ES14.7, 1x, ES9.2, 1x, F7.2, &
        1x, F7.2, A1, 1x, F6.2, 1x, F7.2, 1x, I3)
  if (entry%n_it /= 1) then
    write (u, 1) &
      entry%it, &
      entry%n_calls, &
      entry%integral, &
      abs(entry%error), &
      abs(integration_entry_get_relative_error (entry)) * 100, &
      abs(integration_entry_get_accuracy (entry)), &
      star, &
      entry%efficiency * 100, &
      entry%chi2, &
      entry%n_it
  else
    write (u, 1) &
      entry%it, &
      entry%n_calls, &
      entry%integral, &
      abs(entry%error), &
      abs(integration_entry_get_relative_error (entry)) * 100, &
      abs(integration_entry_get_accuracy (entry)), &
      star, &
      entry%efficiency * 100
  end if
end if
flush (u)
end subroutine integration_entry_write

```

Read the entry, assuming it has been written in verbose format.

*(Integration results: procedures)+≡*

```

subroutine integration_entry_read (entry, unit)
  type(integration_entry_t), intent(out) :: entry
  integer, intent(in) :: unit

```

```

character(30) :: dummy
character :: equals
integer :: n_groves
read (unit, *) dummy, equals, entry%process_type
read (unit, *) dummy, equals, entry%pass
read (unit, *) dummy, equals, entry%it
read (unit, *) dummy, equals, entry%n_it
read (unit, *) dummy, equals, entry%n_calls
read (unit, *) dummy, equals, entry%improved
read (unit, *) dummy, equals, entry%integral
read (unit, *) dummy, equals, entry%error
read (unit, *) dummy, equals, entry%efficiency
read (unit, *) dummy, equals, entry%chi2
read (unit, *) dummy, equals, n_groves
if (n_groves /= 0) then
    allocate (entry%grove_weight (n_groves))
    read (unit, *) dummy, equals, entry%grove_weight
end if
end subroutine integration_entry_read

```

Write an account of the channel weights, accumulated by groves.

*(Integration results: procedures)+≡*

```

subroutine integration_entry_write_grove_weights (entry, unit)
    type(integration_entry_t), intent(in) :: entry
    integer, intent(in), optional :: unit
    integer :: n_groves
    character(20) :: fmt
    integer :: u
    u = output_unit (unit); if (u < 0) return
    if (allocated (entry%grove_weight)) then
        n_groves = size (entry%grove_weight)
        write (fmt, "(A,I0,A)") "(", n_groves, "(1x,I3))"
        write (u, fmt) nint (entry%grove_weight * 100)
    end if
! contains
!     function get_ifmt (n) result (fmt)
!         character(2) :: fmt
!         integer, intent(in) :: n
!         character(20) :: tmp_str
!         integer :: ilen
!         write (tmp_str, "(I0)") n
!         ilen = len_trim (tmp_str)
!         write (fmt, "(A1,I1)") "I", ilen
!     end function get_ifmt
end subroutine integration_entry_write_grove_weights

```

Compute the average for all entries in the specified integration pass. The integrals are weighted w.r.t. their individual errors, and we compute average, error of the average, and  $\chi^2$  value. All errors are assumed Gaussian, of course. The efficiency returned is the one of the last entry in the integration pass. For the time estimate, we simply take the minimum of all which should correspond to the best efficiency.

If any error vanishes, averaging by this algorithm would fail. In this case,

we simply average the entries and use the deviations from this average (if any) to estimate the error.

*(Integration results: procedures)* +=

```
function compute_average (entry, pass) result (result)
  type(integration_entry_t) :: result
  type(integration_entry_t), dimension(:), intent(in) :: entry
  integer, intent(in) :: pass
  integer :: i
  logical, dimension(size(entry)) :: mask
  real(default), dimension(size(entry)) :: ivar
  real(default) :: sum_ivar, variance
  result%process_type = entry(1)%process_type
  result%pass = pass
  mask = entry%pass == pass .and. entry%process_type /= PRC_UNKNOWN
  result%it = maxval (entry%it, mask)
  result%n_it = count (mask)
  result%n_calls = sum (entry%n_calls, mask)
  if (all (entry%error /= 0)) then
    ivar = 1 / entry%error ** 2
    sum_ivar = sum (ivar, mask)
    if (sum_ivar /= 0) then
      variance = 1 / sum_ivar
    else
      variance = 0
    end if
    result%integral = sum (entry%integral * ivar, mask) * variance
  if (result%n_it > 1) then
    result%chi2 = &
      sum ((entry%integral - result%integral)**2 * ivar, mask) &
      / (result%n_it - 1)
  end if
else if (result%n_it /= 0) then
  result%integral = sum (entry%integral, mask) / result%n_it
  if (result%n_it > 1) then
    variance = &
      sum ((entry%integral - result%integral)**2, mask) &
      / (result%n_it - 1)
    if (result%integral /= 0) then
      if (abs (variance / result%integral) &
        < 100 * epsilon (1._default)) then
        variance = 0
      end if
    end if
  end if
else
  variance = 0
end if
result%error = sqrt (variance)
do i = size (entry), 1, -1
  if (mask(i)) then
    result%efficiency = entry(i)%efficiency
    exit
  end if
end do
```

```
end function compute_average
```

### 15.1.2 Combined integration results

We collect a list of results which grows during the execution of the program. This is implemented as an array which grows if necessary; so we can easily compute averages.

We implement this as an extension of the `mci_results_t` which is defined in `mci_base` as an abstract type. We thus decouple the implementation of the integrator from the implementation of the results display, but nevertheless can record intermediate results during integration. This implies that the present extension implements a `record` method.

```
<Integration results: public>≡
  public :: integration_results_t

<Integration results: types>+≡
  type, extends (mci_results_t) :: integration_results_t
    private
    integer :: process_type = PRC_UNKNOWN
    integer :: current_pass = 0
    integer :: n_pass = 0
    integer :: n_it = 0
    logical :: screen = .false.
    integer :: unit = 0
    type(integration_entry_t), dimension(:), allocatable :: entry
    type(integration_entry_t), dimension(:), allocatable :: average
  contains
    <Integration results: integration results: TBP>
  end type integration_results_t
```

The array is extended in chunks of 10 entries.

```
<Integration results: parameters>+≡
  integer, parameter :: RESULTS_CHUNK_SIZE = 10
```

The standard does not require to explicitly initialize the integers; however, some gfortran version has a bug here and misses the default initialization in the type definition.

```
<Integration results: integration results: TBP>≡
  procedure :: init => integration_results_init

<Integration results: procedures>+≡
  subroutine integration_results_init (results, process_type)
    class(integration_results_t), intent(out) :: results
    integer, intent(in) :: process_type
    results%process_type = process_type
    results%n_pass = 0
    results%n_it = 0
    allocate (results%entry (RESULTS_CHUNK_SIZE))
    allocate (results%average (RESULTS_CHUNK_SIZE))
  end subroutine integration_results_init
```



Output (ASCII format). The `verbose` format is used for writing the header in grid files.

*(Integration results: integration results: TBP)*+≡

```
procedure :: write => integration_results_write
```

*(Integration results: procedures)*+≡

```
subroutine integration_results_write (object, unit, verbose)
  class(integration_results_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: u, n
  real(default) :: time_per_event
  u = output_unit (unit); if (u < 0) return
  verb = .false.; if (present (verbose)) verb = verbose
  if (.not. verb) then
    call write_dline (unit)
    if (object%n_it /= 0) then
      call write_header (object%entry(1)%process_type, unit, &
        logfile=.false.)
      call write_dline (unit)
      do n = 1, object%n_it
        if (n > 1) then
          if (object%entry(n)%pass /= object%entry(n-1)%pass) then
            call write_hline (unit)
            call integration_entry_write &
              (object%average(object%entry(n-1)%pass), unit)
            call write_hline (unit)
          end if
        end if
        call integration_entry_write (object%entry(n), unit)
      end do
      call write_hline(unit)
      call integration_entry_write (object%average(object%n_pass), unit)
    else
      call msg_message ("[WHIZARD integration results: empty]", unit)
    end if
    call write_dline (unit)
  else
    write (u, *) "begin(integration_results)"
    write (u, *) "  n_pass = ", object%n_pass
    write (u, *) "  n_it = ", object%n_it
    if (object%n_it > 0) then
      write (u, *) "begin(integration_pass)"
      do n = 1, object%n_it
        if (n > 1) then
          if (object%entry(n)%pass /= object%entry(n-1)%pass) then
            write (u, *) "end(integration_pass)"
            write (u, *) "begin(integration_pass)"
          end if
        end if
        write (u, *) "begin(iteration)"
        call integration_entry_write (object%entry(n), unit, verb)
        write (u, *) "end(iteration)"
      end do
    end if
  end if
end subroutine
```

```

        end do
        write (u, *) "end(integration_pass)"
    end if
    write (u, *) "end(integration_results)"
end if
flush (u)
end subroutine integration_results_write

```

During integration, we do not want to print all results at once, but each intermediate result as soon as we get it. Thus, the previous procedure is chopped in pieces. First piece: store the output unit and a flag whether we want to print to standard output as well. Then write the header if the results are still empty, i.e., before integration has started. The second piece writes a single result to the saved output channels. We call this from the `record` method, which can be called from the integrator directly. The third piece writes the average result, once a pass has been completed. The fourth piece writes a footer (if any), assuming that this is the final result.

*(Integration results: integration results: TBP)+≡*

```

procedure :: display_init => integration_results_display_init
procedure :: display_current => integration_results_display_current
procedure :: display_pass => integration_results_display_pass
procedure :: display_final => integration_results_display_final

```

*(Integration results: procedures)+≡*

```

subroutine integration_results_display_init &
    (results, process_type, screen, unit)
    class(integration_results_t), intent(inout) :: results
    integer, intent(in) :: process_type
    logical, intent(in) :: screen
    integer, intent(in), optional :: unit
    integer :: u
    if (present (unit)) results%unit = unit
    u = output_unit ()
    results%screen = screen
    if (results%n_it == 0) then
        if (results%screen) then
            call write_dline (u)
            call write_header (process_type, u, &
                logfile=.false.)
            call write_dline (u)
        end if
        if (results%unit /= 0) then
            call write_dline (results%unit)
            call write_header (process_type, results%unit, &
                logfile=.false.)
            call write_dline (results%unit)
        end if
    else
        if (results%screen) then
            call write_hline (u)
        end if
        if (results%unit /= 0) then
            call write_hline (results%unit)
        end if
    end if
end subroutine integration_results_display_init

```

```

        end if
    end if
end subroutine integration_results_display_init

subroutine integration_results_display_current (results)
    class(integration_results_t), intent(in) :: results
    integer :: u
    u = output_unit ()
    if (results%screen) then
        call integration_entry_write (results%entry(results%n_it), u)
    end if
    if (results%unit /= 0) then
        call integration_entry_write (results%entry(results%n_it), results%unit)
    end if
end subroutine integration_results_display_current

subroutine integration_results_display_pass (results)
    class(integration_results_t), intent(in) :: results
    integer :: u
    u = output_unit ()
    if (results%screen) then
        call write_hline (u)
        call integration_entry_write &
            (results%average(results%entry(results%n_it)%pass), u)
    end if
    if (results%unit /= 0) then
        call write_hline (results%unit)
        call integration_entry_write &
            (results%average(results%entry(results%n_it)%pass), results%unit)
    end if
end subroutine integration_results_display_pass

subroutine integration_results_display_final (results)
    class(integration_results_t), intent(inout) :: results
    integer :: u
    u = output_unit ()
    if (results%screen) then
        call write_dline (u)
    end if
    if (results%unit /= 0) then
        call write_dline (results%unit)
    end if
    results%screen = .false.
    results%unit = 0
end subroutine integration_results_display_final

```

Incremental output: Write specific / last line. separator if appropriate.

*(Integration results: procedures)* +=

```

subroutine integration_results_write_entry (results, it, unit)
    type(integration_results_t), intent(in) :: results
    integer, intent(in) :: it
    integer, intent(in), optional :: unit
    integer :: u

```

```

    u = output_unit (unit); if (u < 0) return
    if (it /= 0) call integration_entry_write (results%entry(it), unit)
end subroutine integration_results_write_entry

subroutine integration_results_write_current (results, unit)
    type(integration_results_t), intent(in) :: results
    integer, intent(in), optional :: unit
    integer :: u, n
    u = output_unit (unit); if (u < 0) return
    n = results%n_it
    if (n /= 0) call integration_entry_write (results%entry(n), unit)
end subroutine integration_results_write_current

```

Write one line for the average

*(Integration results: procedures)+≡*

```

subroutine integration_results_write_average (results, pass, unit)
    type(integration_results_t), intent(in) :: results
    integer, intent(in) :: pass
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    if (allocated (results%average) .and. pass /= 0) &
        call integration_entry_write (results%average(pass), unit)
end subroutine integration_results_write_average

subroutine integration_results_write_current_average (results, unit)
    type(integration_results_t), intent(in) :: results
    integer, intent(in), optional :: unit
    integer :: u, n
    u = output_unit (unit); if (u < 0) return
    n = results%n_pass
    if (allocated (results%average) .and. n /= 0) &
        call integration_entry_write (results%average(n), unit)
end subroutine integration_results_write_current_average

```

Write a concise table of grove weights, i.e., the channel history.

*(Integration results: procedures)+≡*

```

subroutine integration_results_write_grove_weights (results, unit)
    type(integration_results_t), intent(in) :: results
    integer, intent(in), optional :: unit
    integer :: u, i, n
    u = output_unit (unit); if (u < 0) return
    if (results%n_it /= 0) then
        call msg_message ("Phase-space grove weight history: " &
            // "(numbers in %)", unit)
        write (u, "(A9)", advance="no") "| grove |"
        do i = 1, integration_entry_get_n_groves (results%entry(1))
            write (u, "(1x,I3)", advance="no") i
        end do
        write (u, *)
        call write_dline (unit)
        do n = 1, results%n_it
            if (n > 1) then

```

```

        if (results%entry(n)%pass /= results%entry(n-1)%pass) then
            call write_hline (unit)
        end if
    end if
    write (u, "(1x,I6,1x,A1)", advance="no")  n, "|"
    call integration_entry_write_grove_weights (results%entry(n), unit)
end do
else
    call msg_message ("Channel weight history: [undefined]", unit)
end if
flush (u)
call write_dline(unit)
end subroutine integration_results_write_grove_weights

```

Create a copy. Since all components are implemented as allocatable, intrinsic assignment will do.

```

<XXX Integration results: integration results: TBP>≡
    procedure :: copy_to => integration_results_copy_to

<XXX Integration results: procedures>≡
    subroutine integration_results_copy_to (results, copy)
        class(integration_results_t), intent(in) :: results
        type(integration_results_t), intent(out) :: copy
        copy = results
    end subroutine integration_results_copy_to

```

Read the list from file. The file must be written using the `verbose` option of the writing routine.

```

<Integration results: procedures>+≡
    subroutine integration_results_read (results, unit)
        type(integration_results_t), intent(out) :: results
        integer, intent(in) :: unit
        character(80) :: buffer
        character :: equals
        integer :: pass, it
        read (unit, *) buffer
        if (trim (adjustl (buffer)) /= "begin(integration_results)") then
            call read_err (); return
        end if
        read (unit, *) buffer, equals, results%n_pass
        read (unit, *) buffer, equals, results%n_it
        allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
        allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
        it = 0
        do pass = 1, results%n_pass
            read (unit, *) buffer
            if (trim (adjustl (buffer)) /= "begin(integration_pass)") then
                call read_err (); return
            end if
            READ_ENTRIES: do
                read (unit, *) buffer
                if (trim (adjustl (buffer)) /= "begin(iteration)") then
                    exit READ_ENTRIES
                end if
            end do
            results%entry(it) = buffer
            results%average(it) = 0
            it = it + 1
        end do
    end subroutine integration_results_read

```

```

        end if
        it = it + 1
        call integration_entry_read (results%entry(it), unit)
        read (unit, *) buffer
        if (trim (adjustl (buffer)) /= "end(iteration)") then
            call read_err (); return
        end if
    end do READ_ENTRIES
    if (trim (adjustl (buffer)) /= "end(integration_pass)") then
        call read_err (); return
    end if
    results%average(pass) = compute_average (results%entry, pass)
end do
read (unit, *) buffer
if (trim (adjustl (buffer)) /= "end(integration_results)") then
    call read_err (); return
end if
contains
subroutine read_err ()
    call msg_fatal ("Reading integration results from file: syntax error")
end subroutine read_err
end subroutine integration_results_read

```

Check integration results for consistency. We compare against an array of pass indices and call numbers. If there is a difference, up to the number of iterations done so far, we return failure. Dummy entries (where `pass = 0`) are ignored.

*(Integration results: procedures)*+≡

```

function integration_results_iterations_are_consistent &
    (results, pass, n_calls) result (flag)
    logical :: flag
    type(integration_results_t), intent(in) :: results
    integer, dimension(:), intent(in) :: pass, n_calls
    integer :: n_it
    n_it = results%n_it
    flag = size (pass) >= n_it .and. size (n_calls) >= n_it
    if (flag) then
        flag = all (results%entry(:n_it)%pass == pass(:n_it) &
            .and. &
            (results%entry(:n_it)%n_calls == n_calls(:n_it) &
            .or. &
            results%entry(:n_it)%process_type == PRC_UNKNOWN))
    end if
end function integration_results_iterations_are_consistent

```

Discard all results starting from the specified iteration.

*(Integration results: procedures)*+≡

```

subroutine integration_results_discard (results, it)
    type(integration_results_t), intent(inout) :: results
    integer, intent(in) :: it
    if (it <= results%n_it) then
        select case (it)
        case (:1)
            results%n_it = 0

```

```

        results%n_pass = 0
        results%current_pass = 0
    case default
        results%n_it = it - 1
        results%n_pass = maxval (results%entry(1:results%n_it)%pass)
        results%current_pass = results%n_pass
    end select
end if
end subroutine integration_results_discard

```

Expand the list of entries if the limit has been reached:

```

<Integration results: procedures>+≡
subroutine integration_results_expand (results)
    class(integration_results_t), intent(inout) :: results
    type(integration_entry_t), dimension(:), allocatable :: entry_tmp
    if (results%n_it == size (results%entry)) then
        allocate (entry_tmp (results%n_it))
        entry_tmp = results%entry
        deallocate (results%entry)
        allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
        results%entry(:results%n_it) = entry_tmp
        deallocate (entry_tmp)
    end if
    if (results%n_pass == size (results%average)) then
        allocate (entry_tmp (results%n_pass))
        entry_tmp = results%average
        deallocate (results%average)
        allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
        results%average(:results%n_pass) = entry_tmp
        deallocate (entry_tmp)
    end if
end subroutine integration_results_expand

```

Increment the `current_pass` counter. Can be done before integration; after integration, the recording method may use the value of this counter to define the entry.

```

<Integration results: integration results: TBP>+≡
    procedure :: new_pass => integration_results_new_pass

<Integration results: procedures>+≡
subroutine integration_results_new_pass (results)
    class(integration_results_t), intent(inout) :: results
    results%current_pass = results%current_pass + 1
end subroutine integration_results_new_pass

```

Append a new entry to the list and, if appropriate, compute the average.

```

<Integration results: procedures>+≡
subroutine integration_results_append_entry (results, entry)
    class(integration_results_t), intent(inout) :: results
    type(integration_entry_t), intent(in), optional :: entry
    if (results%n_it == 0) then
        results%n_it = 1
        results%n_pass = 1
    end if
end subroutine integration_results_append_entry

```

```

else
  call integration_results_expand (results)
  if (present (entry)) then
    if (entry%pass /= results%entry(results%n_it)%pass) &
      results%n_pass = results%n_pass + 1
    end if
    results%n_it = results%n_it + 1
  end if
  if (present (entry)) then
    results%entry(results%n_it) = entry
    results%average(results%n_pass) = &
      compute_average (results%entry, entry%pass)
  end if
end subroutine integration_results_append_entry

```

Enter results into the results list.

```

<Integration results: TBP>≡
  procedure :: append => integration_results_append

<Integration results: procedures>+≡
  subroutine integration_results_append (results, &
    n_it, n_calls, &
    integral, error, efficiency, &
    grove_weight, time_start, time_end)
    class(integration_results_t), intent(inout) :: results
    integer, intent(in) :: n_it, n_calls
    real(default), intent(in) :: integral, error, efficiency
    real(default), dimension(:), intent(in), optional :: grove_weight
    type(time_t), intent(in), optional :: time_start, time_end
    logical :: improved
    type(integration_entry_t) :: entry
    if (results%n_it /= 0) then
      improved = abs(accuracy (integral, error, n_calls)) &
        < abs(integration_entry_get_accuracy (results%entry(results%n_it)))
    else
      improved = .true.
    end if
    call integration_entry_init (entry, &
      results%process_type, results%current_pass, &
      results%n_it+1, n_it, n_calls, improved, &
      integral, error, efficiency, &
      grove_weight=grove_weight, time_start=time_start, time_end=time_end)
    call integration_results_append_entry (results, entry)
  end subroutine integration_results_append

```

Enter an empty result into the results list.

```

<Integration results: public>+≡
  public :: integration_results_append_null

<Integration results: procedures>+≡
  subroutine integration_results_append_null (results, pass, n_it)
    type(integration_results_t), intent(inout) :: results
    integer, intent(in) :: pass, n_it
    type(integration_entry_t) :: entry

```



```

    call integration_entry_init (entry, &
        PRC_UNKNOWN, results%current_pass, n_it, 1, 0, .false., &
        0._default, 0._default, 0._default)
    call integration_results_append_entry (results, entry)
end subroutine integration_results_append_null

```

Record an integration pass executed by an mci integrator object.

Note: grove weight, timings not implemented yet.

```

<Integration results: integration results: TBP>+≡
    procedure :: record => integration_results_record

<Integration results: procedures>+≡
    subroutine integration_results_record &
        (object, n_it, n_calls, integral, error, efficiency)
        class(integration_results_t), intent(inout) :: object
        integer, intent(in) :: n_it, n_calls
        real(default), intent(in) :: integral, error, efficiency
        call integration_results_append (object, &
            n_it, n_calls, &
            integral, error, efficiency)
        call object%display_current ()
    end subroutine integration_results_record

```

### 15.1.3 Access results

Return true if the results object has entries.

```

<Integration results: integration results: TBP>+≡
    procedure :: exist => integration_results_exist

<Integration results: procedures>+≡
    function integration_results_exist (results) result (flag)
        logical :: flag
        class(integration_results_t), intent(in) :: results
        flag = results%n_pass > 0
    end function integration_results_exist

```

Retrieve information from the results record. If **last** is set and true, take the last iteration. If **it** is set instead, take this iteration. If **pass** is set, take this average. If none is set, take the final average.

If the result would be invalid, the entry is not assigned. Due to default initialization, this returns a null entry.

```

<Integration results: integration results: TBP>+≡
    procedure :: get_entry => results_get_entry

<Integration results: procedures>+≡
    function results_get_entry (results, last, it, pass) result (entry)
        class(integration_results_t), intent(in) :: results
        type(integration_entry_t) :: entry
        logical, intent(in), optional :: last
        integer, intent(in), optional :: it, pass
        if (present (last)) then
            if (allocated (results%entry) .and. results%n_it > 0) then

```

```

        entry = results%entry(results%n_it)
    else
        call error ()
    end if
else if (present (it)) then
    if (allocated (results%entry) .and. it > 0 .and. it <= results%n_it) then
        entry = results%entry(it)
    else
        call error ()
    end if
else if (present (pass)) then
    if (allocated (results%average) &
        .and. pass > 0 .and. pass <= results%n_pass) then
        entry = results%average (pass)
    else
        call error ()
    end if
else
    if (allocated (results%average) .and. results%n_pass > 0) then
        entry = results%average (results%n_pass)
    else
        call error ()
    end if
end if
contains
    subroutine error ()
        call msg_fatal ("Requested integration result is not available")
    end subroutine error
end function results_get_entry

```

The individual procedures. The `results` record should have the `target` attribute, but only locally within the function.

*(Integration results: integration results: TBP)+≡*

```

procedure :: get_n_calls => integration_results_get_n_calls
procedure :: get_integral => integration_results_get_integral
procedure :: get_error => integration_results_get_error
procedure :: get_accuracy => integration_results_get_accuracy
procedure :: get_efficiency => integration_results_get_efficiency

```

*(Integration results: procedures)+≡*

```

function integration_results_get_n_calls (results, last, it, pass) &
    result (n_calls)
    class(integration_results_t), intent(in), target :: results
    integer :: n_calls
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    n_calls = integration_entry_get_n_calls &
        (results%get_entry (last, it, pass))
end function integration_results_get_n_calls

function integration_results_get_integral (results, last, it, pass) &
    result (integral)
    class(integration_results_t), intent(in), target :: results
    real(default) :: integral

```

```

    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    integral = integration_entry_get_integral &
        (results%get_entry (last, it, pass))
end function integration_results_get_integral

function integration_results_get_error (results, last, it, pass) &
    result (error)
    class(integration_results_t), intent(in), target :: results
    real(default) :: error
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    error = integration_entry_get_error &
        (results%get_entry (last, it, pass))
end function integration_results_get_error

function integration_results_get_accuracy (results, last, it, pass) &
    result (accuracy)
    class(integration_results_t), intent(in), target :: results
    real(default) :: accuracy
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    accuracy = integration_entry_get_accuracy &
        (results%get_entry (last, it, pass))
end function integration_results_get_accuracy

function integration_results_get_chi2 (results, last, it, pass) &
    result (chi2)
    class(integration_results_t), intent(in), target :: results
    real(default) :: chi2
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    chi2 = integration_entry_get_chi2 &
        (results%get_entry (last, it, pass))
end function integration_results_get_chi2

function integration_results_get_efficiency (results, last, it, pass) &
    result (efficiency)
    class(integration_results_t), intent(in), target :: results
    real(default) :: efficiency
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    efficiency = integration_entry_get_efficiency &
        (results%get_entry (last, it, pass))
end function integration_results_get_efficiency

```

Return the time per event for the last integration iteration. (Not for the average.)

*(Integration results: procedures)* +=

```

function integration_results_get_time_per_event (results) result (s)
    real(default) :: s
    type(integration_results_t), intent(in) :: results
    if (results%n_pass /= 0) then

```

```

        s = integration_entry_get_time_per_event (results%entry(results%n_it))
    else
        s = 0
    end if
end function integration_results_get_time_per_event

```

Return the last pass index and the index of the last iteration *within* the last pass. The third routine returns the absolute index of the last iteration.

*(Integration results: procedures)+≡*

```

function integration_results_get_current_pass (results) result (pass)
    integer :: pass
    type(integration_results_t), intent(in) :: results
    pass = results%n_pass
end function integration_results_get_current_pass

function integration_results_get_current_it (results) result (it)
    integer :: it
    type(integration_results_t), intent(in) :: results
    if (allocated (results%entry)) then
        it = count (results%entry(1:results%n_it)%pass == results%n_pass)
    else
        it = 0
    end if
end function integration_results_get_current_it

function integration_results_get_last_it (results) result (it)
    integer :: it
    type(integration_results_t), intent(in) :: results
    it = results%n_it
end function integration_results_get_last_it

```

Return the index of the best iteration (lowest accuracy value) within the current pass. If none qualifies, return zero.

*(Integration results: procedures)+≡*

```

function integration_results_get_best_it (results) result (it)
    integer :: it
    type(integration_results_t), intent(in) :: results
    integer :: i
    real(default) :: acc, acc_best
    acc_best = -1
    it = 0
    do i = 1, results%n_it
        if (results%entry(i)%pass == results%n_pass) then
            acc = integration_entry_get_accuracy (results%entry(i))
            if (acc_best < 0 .or. acc <= acc_best) then
                acc_best = acc
                it = i
            end if
        end if
    end do
end function integration_results_get_best_it

```

Compute the MD5 sum by printing everything and checksumming the resulting file.

```

<Integration results: procedures>+=
function integration_results_get_md5sum (results) result (md5sum_results)
  character(32) :: md5sum_results
  type(integration_results_t), intent(in) :: results
  integer :: u
  u = free_unit ()
  open (unit = u, status = "scratch", action = "readwrite")
  call integration_results_write (results, u, verbose=.true.)
  rewind (u)
  md5sum_results = md5sum (u)
  close (u)
end function integration_results_get_md5sum

```

### 15.1.4 Results display

Write a driver file for history visualization.

The ratio of  $y$  range over  $y$  value must not become too small, otherwise we run into an arithmetic overflow in GAMELAN. 2% appears to be safe.

```

<Limits: public parameters>+=
real, parameter, public :: GML_MIN_RANGE_RATIO = 0.02

<Integration results: procedures>+=
subroutine integration_results_write_driver (results, filename)
  type(integration_results_t), intent(in) :: results
  type(string_t), intent(in) :: filename
  type(string_t) :: file_tex
  integer :: unit
  integer :: n, i, n_pass, pass
  integer, dimension(:), allocatable :: ipass
  real(default) :: ymin, ymax, yavg, ydif, y0, y1
  real(default) :: int, err
  file_tex = filename // ".tex"
  unit = free_unit ()
  open (unit=unit, file=char(file_tex), action="write", status="replace")
  n = results%n_it
  n_pass = results%n_pass
  allocate (ipass (results%n_pass))
  ipass(1) = 0
  pass = 2
  do i = 1, n-1
    if (integration_entry_get_pass (results%entry(i)) &
      /= integration_entry_get_pass (results%entry(i+1))) then
      ipass(pass) = i
      pass = pass + 1
    end if
  end do
  ymin = minval (integration_entry_get_integral (results%entry(:n)) &
    - integration_entry_get_error (results%entry(:n)))
  ymax = maxval (integration_entry_get_integral (results%entry(:n)) &
    + integration_entry_get_error (results%entry(:n)))

```

```

yavg = (ymax + ymin) / 2
ydif = (ymax - ymin)
if (ydif * 1.5 > GML_MIN_RANGE_RATIO * yavg) then
    y0 = yavg - ydif * 0.75
    y1 = yavg + ydif * 0.75
else
    y0 = yavg * (1 - GML_MIN_RANGE_RATIO / 2)
    y1 = yavg * (1 + GML_MIN_RANGE_RATIO / 2)
end if
write (unit, "(A)") "\documentclass{article}"
write (unit, "(A)") "\usepackage{a4wide}"
write (unit, "(A)") "\usepackage{gamelan}"
write (unit, "(A)") "\usepackage{amsmath}"
write (unit, "(A)") ""
write (unit, "(A)") "\begin{document}"
write (unit, "(A)") "\begin{gmlfile}"
write (unit, "(A)") "\section*{Integration Results Display}"
write (unit, "(A)") ""
write (unit, "(A)") "Process: \verb|" // char (filename) // "|"
write (unit, "(A)") ""
write (unit, "(A)") "\vspace*{2\baselineskip}"
write (unit, "(A)") "\unitlength 1mm"
write (unit, "(A)") "\begin{gmlcode}"
write (unit, "(A)") " picture sym; sym = fshape (circle scaled 1mm());"
write (unit, "(A)") " color col.band; col.band = 0.9white;"
write (unit, "(A)") " color col.eband; col.eband = 0.98white;"
write (unit, "(A)") "\end{gmlcode}"
write (unit, "(A)") "\begin{gmlgraph*}(130,180)[history]"
write (unit, "(A)") " setup (linear, linear);"
write (unit, "(A,I0,A)") " history.n_pass = ", n_pass, ";"
write (unit, "(A,I0,A)") " history.n_it = ", n, ";"
write (unit, "(A,A,A)") " history.y0 = #""", char (mp_format (y0)), """,;"
write (unit, "(A,A,A)") " history.y1 = #""", char (mp_format (y1)), """,;"
write (unit, "(A)") &
    " graphrange (#0.5, history.y0), (#(n+0.5), history.y1);"
do pass = 1, n_pass
    write (unit, "(A,I0,A,I0,A)") &
        " history.pass[" , pass, "]" = ", ipass(pass), ";"
    write (unit, "(A,I0,A,A,A)") &
        " history.avg[" , pass, "]" = #""", &
        char (mp_format &
            (integration_entry_get_integral (results%average(pass)))), &
        """,;"
    write (unit, "(A,I0,A,A,A)") &
        " history.err[" , pass, "]" = #""", &
        char (mp_format &
            (integration_entry_get_error (results%average(pass)))), &
        """,;"
    write (unit, "(A,I0,A,A,A)") &
        " history.chi[" , pass, "]" = #""", &
        char (mp_format &
            (integration_entry_get_chi2 (results%average(pass)))), &
        """,;"
end do

```

```

write (unit, "(A,I0,A,I0,A)") &
    " history.pass[" , n_pass + 1, "] = " , n, ";"
write (unit, "(A)") " for i = 1 upto history.n_pass:"
write (unit, "(A)") " if history.chi[i] greater one:"
write (unit, "(A)") " fill plot ("
write (unit, "(A)") &
    " (#(history.pass[i] +.5), " &
    // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
    " (#(history.pass[i+1]+.5), " &
    // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
    " (#(history.pass[i+1]+.5), " &
    // "history.avg[i] plus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
    " (#(history.pass[i] +.5), " &
    // "history.avg[i] plus history.err[i] times history.chi[i])"
write (unit, "(A)") " ) withcolor col.eband fi;"
write (unit, "(A)") " fill plot ("
write (unit, "(A)") &
    " (#(history.pass[i] +.5), history.avg[i] minus history.err[i]),"
write (unit, "(A)") &
    " (#(history.pass[i+1]+.5), history.avg[i] minus history.err[i]),"
write (unit, "(A)") &
    " (#(history.pass[i+1]+.5), history.avg[i] plus history.err[i]),"
write (unit, "(A)") &
    " (#(history.pass[i] +.5), history.avg[i] plus history.err[i])"
write (unit, "(A)") " ) withcolor col.band;"
write (unit, "(A)") " draw plot ("
write (unit, "(A)") &
    " (#(history.pass[i] +.5), history.avg[i]),"
write (unit, "(A)") &
    " (#(history.pass[i+1]+.5), history.avg[i])"
write (unit, "(A)") " ) dashed evenly;"
write (unit, "(A)") " endfor"
write (unit, "(A)") " for i = 1 upto history.n_pass + 1:"
write (unit, "(A)") " draw plot ("
write (unit, "(A)") &
    " (#(history.pass[i]+.5), history.y0),"
write (unit, "(A)") &
    " (#(history.pass[i]+.5), history.y1)"
write (unit, "(A)") " ) dashed withdots;"
write (unit, "(A)") " endfor"
do i = 1, n
    write (unit, "(A,I0,A,A,A,A)") " plot (history) (#", &
        i, ", #""", &
        char (mp_format (integration_entry_get_integral (results%entry(i))), &
        """) vbar #""", &
        char (mp_format (integration_entry_get_error (results%entry(i))), &
        """);"
end do
write (unit, "(A)") " draw piecewise from (history) " &
    // "withsymbol sym;"
write (unit, "(A)") " fullgrid.lr (5,20);"

```

```

write (unit, "(A)" " standardgrid.bt (n);"
write (unit, "(A)" "\end{gmlgraph*}"
write (unit, "(A)" "\end{gmlfile}"
write (unit, "(A)" "\clearpage"
write (unit, "(A)" "\begin{verbatim}"
call integration_results_write (results, unit)
write (unit, "(A)" "\end{verbatim}"
write (unit, "(A)" "\end{document}"
close (unit)
end subroutine integration_results_write_driver

```

Call L<sup>A</sup>T<sub>E</sub>X and Metapost for the history driver file, and convert to PS and PDF.

*(Integration results: procedures)*+≡

```

subroutine integration_results_compile_driver (results, filename, os_data)
  type(integration_results_t), intent(in) :: results
  type(string_t), intent(in) :: filename
  type(os_data_t), intent(in) :: os_data
  integer :: unit, unit_dev, status
  type(string_t) :: file_tex, file_dvi, file_ps, file_pdf, file_mp
  type(string_t) :: setenv_tex, setenv_mp, pipe, pipe_dvi
  type(string_t) :: latex_opt, mpost_opt
  if (.not. os_data%event_analysis) then
    call msg_warning ("Skipping integration history display " &
      // "because latex or mpost is not available")
    return
  end if
  file_tex = filename // ".tex"
  file_dvi = filename // ".dvi"
  file_ps = filename // ".ps"
  file_pdf = filename // ".pdf"
  file_mp = filename // ".mp"
  call msg_message ("Creating integration history display "&
    // char (file_ps) // " and " // char (file_pdf))
  BLOCK: do
    unit_dev = free_unit ()
    open (file = "/dev/null", unit = unit_dev, &
      action = "write", iostat = status)
    if (status /= 0) then
      pipe = ""
      pipe_dvi = ""
    else
      pipe = " > /dev/null"
      pipe_dvi = " 2>/dev/null 1>/dev/null"
    end if
    close (unit_dev)
    if (os_data%whizard_texpath /= "") then
      setenv_tex = &
        "TEXINPUTS=" // os_data%whizard_texpath // " :$TEXINPUTS "
      setenv_mp = &
        "MPINPUTS=" // os_data%whizard_texpath // " :$MPINPUTS "
    else
      setenv_tex = ""
      setenv_mp = ""
    end if
  end do

```



```

end if
call os_system_call (setenv_tex // os_data%latex // " " // &
    file_tex // pipe, status)
if (status /= 0) exit BLOCK
if (os_data%gml /= "") then
    call os_system_call (setenv_mp // os_data%gml // " " // &
        file_mp // pipe, status)
else
    call msg_error ("Could not use GAMELAN/MetaPOST.")
    exit BLOCK
end if
if (status /= 0) exit BLOCK
call os_system_call (setenv_tex // os_data%latex // " " // &
    file_tex // pipe, status)
if (status /= 0) exit BLOCK
if (os_data%event_analysis_ps) then
    call os_system_call (os_data%dvips // " " // &
        file_dvi // pipe_dvi, status)
    if (status /= 0) exit BLOCK
else
    call msg_warning ("Skipping PostScript generation because dvips " &
        // "is not available")
    exit BLOCK
end if
if (os_data%event_analysis_pdf) then
    call os_system_call (os_data%ps2pdf // " " // &
        file_ps, status)
    if (status /= 0) exit BLOCK
else
    call msg_warning ("Skipping PDF generation because ps2pdf " &
        // "is not available")
    exit BLOCK
end if
exit BLOCK
end do BLOCK
if (status /= 0) then
    call msg_error ("Unable to compile integration history display")
end if
end subroutine integration_results_compile_driver

```

## 15.2 Abstract process core

In this module we provide abstract data types for process classes. Each process class represents a set of processes which are handled by a common “method”, e.g., by the O’MEGA matrix-element generator. The process class is also able to select a particular implementation for the phase-space and integration modules.

For a complete implementation of a process class, we have to provide extensions of the following abstract types:

**prc\_core\_def\_t** process and matrix-element configuration

**prc\_writer\_t** (optional) writing external matrix-element code

**prc\_driver\_t** accessing the matrix element (internal or external)

**prc\_core\_t** evaluating kinematics and matrix element. The process core also selects phase-space and integrator implementations as appropriate for the process class and configuration.

In the actual process-handling data structures, each process component contains an instance of such a process class as its core. This allows us to keep the **processes** module below, which supervises matrix-element evaluation, integration, and event generation, free of any reference to concrete implementations (for the process class, phase space, and integrator).

There are no unit tests, these are deferred to the **processes** module.

```
(prc_core.f90)≡
  <File header>
  module prc_core

    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use interactions

    use sf_base
    use process_constants
    use mci_base

    use prc_core_def
    use process_libraries

    <Standard module head>

    <Prc core: public>

    <Prc core: types>

    <Prc core: interfaces>

    contains

    <Prc core: procedures>

  end module prc_core
```

### 15.2.1 The process core

The process core is of abstract data type. Different types of matrix element will be represented by different implementations.

```
<Prc core: public>≡
  public :: prc_core_t

<Prc core: types>≡
  type, abstract :: prc_core_t
```

```

class(prc_core_def_t), pointer :: def => null ()
logical :: data_known = .false.
type(process_constants_t) :: data
class(prc_core_driver_t), allocatable :: driver
contains
  <Prc core: process core: TBP>
end type prc_core_t

```

In any case there must be an output routine.

```

<Prc core: process core: TBP>≡
  procedure(prc_core_write), deferred :: write

<Prc core: interfaces>≡
  abstract interface
    subroutine prc_core_write (object, unit)
      import
      class(prc_core_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine prc_core_write
  end interface

```

For initialization, we assign a pointer to the process entry in the relevant library. This allows us to access all process functions via the implementation of `prc_core_t`.

We declare the object as `intent(inout)`, since just after allocation it may be useful to store some extra data in the object, which we can then use in the actual initialization. This applies to extensions of `prc_core` which override the `init` method.

```

<Prc core: process core: TBP>+≡
  procedure :: init => prc_core_init
  procedure :: base_init => prc_core_init

<Prc core: procedures>≡
  subroutine prc_core_init (object, def, lib, id, i_component)
    class(prc_core_t), intent(inout) :: object
    class(prc_core_def_t), intent(in), target :: def
    type(process_library_t), intent(in), target :: lib
    type(string_t), intent(in) :: id
    integer, intent(in) :: i_component
    object%def => def
    call lib%connect_process (id, i_component, object%data, object%driver)
    object%data_known = .true.
  end subroutine prc_core_init

```

Return true if a MC dataset should be attached to this process component. False if it shares the dataset with another component.

```

<Prc core: process core: TBP>+≡
  procedure(prc_core_get_flag), deferred :: needs_mcset

<Prc core: interfaces>+≡
  abstract interface
    function prc_core_get_flag (object) result (flag)
      import

```

```

        class(prc_core_t), intent(in) :: object
        logical :: flag
    end function prc_core_get_flag
end interface

```

Return an integer number

```

<Prc core: interfaces>+≡
    abstract interface
        function prc_core_get_integer (object) result (i)
            import
            class(prc_core_t), intent(in) :: object
            integer :: i
        end function prc_core_get_integer
    end interface

```

Return the number of distinct terms requested by this process component.

```

<Prc core: process core: TBP>+≡
    procedure(prc_core_get_integer), deferred :: get_n_terms

```

Tell whether a particular combination of flavor/helicity/color state is allowed for the matrix element.

```

<Prc core: process core: TBP>+≡
    procedure(prc_core_is_allowed), deferred :: is_allowed
<Prc core: interfaces>+≡
    abstract interface
        function prc_core_is_allowed (object, i_term, f, h, c) result (flag)
            import
            class(prc_core_t), intent(in) :: object
            integer, intent(in) :: i_term, f, h, c
            logical :: flag
        end function prc_core_is_allowed
    end interface

```

Set the constant process data for a specific term. By default, these are the constants stored inside the object, ignoring the term index. Type extensions may override this and provide term-specific data.

```

<Prc core: process core: TBP>+≡
    procedure :: get_constants => prc_core_get_constants
<Prc core: procedures>+≡
    subroutine prc_core_get_constants (object, data, i_term)
        class(prc_core_t), intent(in) :: object
        type(process_constants_t), intent(out) :: data
        integer, intent(in) :: i_term
        data = object%data
    end subroutine prc_core_get_constants

```

The strong coupling is not among the process constants. The default implementation is to return a negative number, which indicates that  $\alpha_s$  is not available. This may be overridden by an implementation that provides an (event-specific) value. The value can be stored in the `tmp` workspace.

```

<Prc core: process core: TBP>+≡

```

```

    procedure :: get_alpha_s => prc_core_get_alpha_s
  <Prc core: procedures>+≡
    function prc_core_get_alpha_s (object, tmp) result (alpha)
      class(prc_core_t), intent(in) :: object
      class(workspace_t), intent(in), allocatable :: tmp
      real(default) :: alpha
      alpha = -1
    end function prc_core_get_alpha_s

```

Allocate the workspace associated to a process component. The default is that there is no workspace, so we do nothing. A type extension may override this and allocate a workspace object of appropriate type, which can be used in further calculations.

In any case, the `intent(out)` attribute deletes any previously allocated workspace.

```

  <Prc core: process core: TBP>+≡
    procedure :: allocate_workspace => prc_core_ignore_workspace
  <Prc core: procedures>+≡
    subroutine prc_core_ignore_workspace (object, tmp)
      class(prc_core_t), intent(in) :: object
      class(workspace_t), intent(inout), allocatable :: tmp
    end subroutine prc_core_ignore_workspace

```

Initialize the structure-function instance that corresponds to a process component. In ordinary cases, this amounts to a straightforward copy of the given template, but the process core may also choose to modify the setup, and it may access its own workspace.

The `sf_chain_instance` is the object that we want to prepare. The `sf_chain` argument is a template for its structure. For the initialization, we also need the number of channels `n_channel`.

Note: crash with nagfor 5.3.1 if `sf_chain_instance` is declared `intent(out)`, as would be more appropriate.

```

  <Prc core: process core: TBP>+≡
    procedure :: init_sf_chain => prc_core_init_sf_chain
  <Prc core: procedures>+≡
    subroutine prc_core_init_sf_chain &
      (object, sf_chain_instance, sf_chain, n_channel, tmp)
      class(prc_core_t), intent(in) :: object
      type(sf_chain_instance_t), intent(inout), target :: sf_chain_instance
      type(sf_chain_t), intent(in), target :: sf_chain
      integer, intent(in) :: n_channel
      class(workspace_t), intent(inout), allocatable :: tmp
      call sf_chain_instance%init (sf_chain, n_channel)
    end subroutine prc_core_init_sf_chain

```

Compute the momenta in the hard interaction, taking the seed kinematics as input. The `i_term` index tells us which term we want to compute. (The standard method is to just transfer the momenta to the hard interaction.)

```

  <Prc core: process core: TBP>+≡

```

```

    procedure(prc_core_compute_hard_kinematics), deferred :: &
        compute_hard_kinematics
<Prc core: interfaces>+≡
    abstract interface
        subroutine prc_core_compute_hard_kinematics &
            (object, p_seed, i_term, int_hard, tmp)
        import
            class(prc_core_t), intent(in) :: object
            type(vector4_t), dimension(:), intent(in) :: p_seed
            integer, intent(in) :: i_term
            type(interaction_t), intent(inout) :: int_hard
            class(workspace_t), intent(inout), allocatable :: tmp
        end subroutine prc_core_compute_hard_kinematics
    end interface

```

Compute the momenta in the effective interaction, taking the hard kinematics as input. (This is called only if parton recombination is to be applied for the process variant.)

```

<Prc core: process core: TBP>+≡
    procedure(prc_core_compute_eff_kinematics), deferred :: &
        compute_eff_kinematics
<Prc core: interfaces>+≡
    abstract interface
        subroutine prc_core_compute_eff_kinematics &
            (object, i_term, int_hard, int_eff, tmp)
        import
            class(prc_core_t), intent(in) :: object
            integer, intent(in) :: i_term
            type(interaction_t), intent(in) :: int_hard
            type(interaction_t), intent(inout) :: int_eff
            class(workspace_t), intent(inout), allocatable :: tmp
        end subroutine prc_core_compute_eff_kinematics
    end interface

```

Recover the missing pieces. We know the incoming momenta of the `p_seed` array and the outgoing momenta of the `int_eff` interaction. We have to recover the outgoing momenta of `p_seed` and the incoming momenta of `int_eff`.

(The trivial case is that these are identical.)

Furthermore, if `int_hard` and `int_eff` are not aliased, we have to set the momenta there. In the trivial case, `int_eff` is a pointer to `int_hard`, so `int_hard` should not be touched at all.

```

<Prc core: process core: TBP>+≡
    procedure(prc_core_recover_kinematics), deferred :: &
        recover_kinematics
<Prc core: interfaces>+≡
    abstract interface
        subroutine prc_core_recover_kinematics &
            (object, p_seed, int_hard, int_eff, tmp)
        import
            class(prc_core_t), intent(in) :: object
            type(vector4_t), dimension(:), intent(inout) :: p_seed

```

```

        type(interaction_t), intent(inout) :: int_hard
        type(interaction_t), intent(inout) :: int_eff
        class(workspace_t), intent(inout), allocatable :: tmp
    end subroutine prc_core_recover_kinematics
end interface

```

The process core must implement this function. Here,  $j$  is the index of the particular term we want to compute. The amplitude may depend on the factorization and renormalization scales.

The `tmp` (workspace) argument may be used if it is provided by the caller. Otherwise, the routine should compute the result directly.

```

<Prc core: process core: TBP>+≡
    procedure(prc_core_compute_amplitude), deferred :: compute_amplitude

<Prc core: interfaces>+≡
    abstract interface
        function prc_core_compute_amplitude &
            (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
            import
            class(prc_core_t), intent(in) :: object
            integer, intent(in) :: j
            type(vector4_t), dimension(:), intent(in) :: p
            integer, intent(in) :: f, h, c
            real(default), intent(in) :: fac_scale, ren_scale
            class(workspace_t), intent(inout), allocatable, optional :: tmp
            complex(default) :: amp
        end function prc_core_compute_amplitude
    end interface

```

### 15.2.2 Storage for intermediate results

The abstract `workspace_t` type allows process cores to set up temporary workspace. The object is an extra argument for each of the individual calculations between kinematics setup and matrix-element evaluation.

```

<Prc core: public>+≡
    public :: workspace_t

<Prc core: types>+≡
    type, abstract :: workspace_t
    contains
        procedure(workspace_write), deferred :: write
    end type workspace_t

```

For debugging, we should at least have an output routine.

```

<Prc core: interfaces>+≡
    abstract interface
        subroutine workspace_write (object, unit)
            import
            class(workspace_t), intent(in) :: object
            integer, intent(in), optional :: unit
        end subroutine workspace_write
    end interface

```

```
end interface
```

### 15.2.3 Helicity selection data

This is intended for use with O'MEGA, but may also be made available to other process methods. We set thresholds for counting the times a specific helicity amplitude is zero. When the threshold is reached, we skip this amplitude in subsequent calls.

For initializing the helicity counters, we need an object that holds the two parameters, the threshold (large real number) and the cutoff (integer).

A helicity value suppressed by more than `threshold` (a value which multiplies `epsilon`, to be compared with the average of the current amplitude, default is  $10^{10}$ ) is treated as zero. A matrix element is assumed to be zero and not called again if it has been zero `cutoff` times.

```
<Prc core: public>+≡
    public :: helicity_selection_t

<Prc core: types>+≡
    type :: helicity_selection_t
        logical :: active = .false.
        real(default) :: threshold = 0
        integer :: cutoff = 0
    contains
        <Prc core: helicity selection: TBP>
    end type helicity_selection_t
```

Output. If the selection is inactive, print nothing.

```
<Prc core: helicity selection: TBP>≡
    procedure :: write => helicity_selection_write

<Prc core: procedures>+≡
    subroutine helicity_selection_write (object, unit)
        class(helicity_selection_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        if (object%active) then
            write (u, "(3x,A)") "Helicity selection data:"
            write (u, "(5x,A,ES17.10)") &
                "threshold =", object%threshold
            write (u, "(5x,A,I0)") &
                "cutoff    = ", object%cutoff
        end if
    end subroutine helicity_selection_write
```

## 15.3 Process observables

We define an abstract `subevt_expr_t` object as an extension of the `subevt_t` type. The object contains a local variable list, variable instances (as targets



for pointers in the variable list), and evaluation trees. The evaluation trees reference both the variables and the `subevt`.

There are two instances of the abstract type: one for process instances, one for physical events. Both have a common logical expression `selection` which determines whether the object passes user-defined cuts.

The intention is that we fill the `subevt_t` base object and compute the variables once we have evaluated a kinematical phase space point (or a complete event). We then evaluate the expressions and can use the results in further calculations.

The `process_expr_t` extension contains furthermore scale and weight expressions. The `event_expr_t` extension contains a reweighting-factor expression and a logical expression for event analysis. In practice, we will link the variable list of the `event_obs` object to the variable list of the currently active `process_obs` object, such that the process variables are available to both objects. Event variables are meaningful only for physical events.

```
<subevt_expr.f90>≡
  <File header>
  module subevt_expr

    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use unit_tests
    use os_interface

    use ifiles
    use lexers
    use parser

    use lorentz !NODEP!
    use subevents
    use variables
    use expressions
    use models
    use flavors
    use interactions
    use particles

    <Standard module head>

    <Subevt expr: public>

    <Subevt expr: types>

    <Subevt expr: interfaces>

    contains

    <Subevt expr: procedures>

    <Subevt expr: tests>
```

```
end module subevt_expr
```

### 15.3.1 Abstract base type

```
⟨Subevt expr: types⟩≡
  type, extends (subevt_t), abstract :: subevt_expr_t
    logical :: subevt_filled = .false.
    type(var_list_t) :: var_list
    real(default) :: sqrts_hat = 0
    integer :: n_in = 0
    integer :: n_out = 0
    integer :: n_tot = 0
    logical :: has_selection = .false.
    type(eval_tree_t) :: selection
  contains
    ⟨Subevt expr: subevt expr: TBP⟩
  end type subevt_expr_t
```

Output: Base and extended version. We already have a `write` routine for the `subevt_t` parent type.

```
⟨Subevt expr: subevt expr: TBP⟩≡
  procedure :: base_write => subevt_expr_write

⟨Subevt expr: procedures⟩≡
  subroutine subevt_expr_write (object, unit)
    class(subevt_expr_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Local variables:"
    call write_separator (u)
    call var_list_write (object%var_list, u, follow_link=.false.)
    call write_separator (u)
    if (object%subevt_filled) then
      call object%subevt_t%write (u)
      if (object%has_selection) then
        call write_separator (u)
        write (u, "(1x,A)") "Selection expression:"
        call write_separator (u)
        call eval_tree_write (object%selection, u)
      end if
    else
      write (u, "(1x,A)") "subevt: [undefined]"
    end if
  end subroutine subevt_expr_write
```

Finalizer.

```
⟨Subevt expr: subevt expr: TBP⟩+≡
  procedure (subevt_expr_final), deferred :: final
  procedure :: base_final => subevt_expr_final
```

```

<Subvt expr: procedures>+≡
  subroutine subvt_expr_final (object)
    class(subvt_expr_t), intent(inout) :: object
    call var_list_final (object%var_list)
    if (object%has_selection) then
      call eval_tree_final (object%selection)
    end if
  end subroutine subvt_expr_final

```

### 15.3.2 Initialization

Initialization: define local variables and establish pointers.

The common variables are `sqrts` (the nominal beam energy, fixed), `sqrts_hat` (the actual energy), `n_in`, `n_out`, and `n_tot` for the `subvt`. With the exception of `sqrts`, all are implemented as pointers to subobjects.

```

<Subvt expr: subvt expr: TBP>+≡
  procedure (subvt_expr_setup_vars), deferred :: setup_vars
  procedure :: base_setup_vars => subvt_expr_setup_vars

<Subvt expr: procedures>+≡
  subroutine subvt_expr_setup_vars (expr, sqrts)
    class(subvt_expr_t), intent(inout), target :: expr
    real(default), intent(in) :: sqrts
    call var_list_final (expr%var_list)
    call var_list_append_real (expr%var_list, &
      var_str ("sqrts"), sqrts, &
      locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_real_ptr (expr%var_list, &
      var_str ("sqrts_hat"), expr%sqrts_hat, &
      is_known = expr%subvt_filled, &
      locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_int_ptr (expr%var_list, &
      var_str ("n_in"), expr%n_in, &
      is_known = expr%subvt_filled, &
      locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_int_ptr (expr%var_list, &
      var_str ("n_out"), expr%n_out, &
      is_known = expr%subvt_filled, &
      locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_int_ptr (expr%var_list, &
      var_str ("n_tot"), expr%n_tot, &
      is_known = expr%subvt_filled, &
      locked = .true., verbose = .false., intrinsic = .true.)
  end subroutine subvt_expr_setup_vars

```

Link a variable list to the local one. This could be done event by event, but before evaluating expressions.

```

<Subvt expr: subvt expr: TBP>+≡
  procedure :: link_var_list => subvt_expr_link_var_list

```

```

<Subvt expr: procedures>+≡
  subroutine subvt_expr_link_var_list (expr, var_list)
    class(subvt_expr_t), intent(inout) :: expr
    type(var_list_t), intent(in), target :: var_list
    call var_list_link (expr%var_list, var_list)
  end subroutine subvt_expr_link_var_list

```

Compile the selection expression. If the pointer is disassociated, there is no expression.

```

<Subvt expr: subvt expr: TBP>+≡
  procedure :: setup_selection => subvt_expr_setup_selection

<Subvt expr: procedures>+≡
  subroutine subvt_expr_setup_selection (expr, pn_selection)
    class(subvt_expr_t), intent(inout), target :: expr
    type(parse_node_t), intent(in), pointer :: pn_selection
    if (associated (pn_selection)) then
      call eval_tree_init_lexpr (expr%selection, &
        pn_selection, expr%var_list, expr%subvt_t)
      expr%has_selection = .true.
    end if
  end subroutine subvt_expr_setup_selection

```

### 15.3.3 Evaluation

Reset to initial state, i.e., mark the subvt as invalid.

```

<Subvt expr: subvt expr: TBP>+≡
  procedure :: reset => subvt_expr_reset

<Subvt expr: procedures>+≡
  subroutine subvt_expr_reset (expr)
    class(subvt_expr_t), intent(inout) :: expr
    expr%subvt_filled = .false.
  end subroutine subvt_expr_reset

```

Evaluate the selection expression and return the result. There is also a deferred version: this should evaluate the remaining expressions if the event has passed.

```

<Subvt expr: subvt expr: TBP>+≡
  procedure :: base_evaluate => subvt_expr_evaluate

<Subvt expr: procedures>+≡
  subroutine subvt_expr_evaluate (expr, passed)
    class(subvt_expr_t), intent(inout) :: expr
    logical, intent(out) :: passed
    if (expr%has_selection) then
      call eval_tree_evaluate (expr%selection)
      if (eval_tree_result_is_known (expr%selection)) then
        passed = eval_tree_get_log (expr%selection)
      else
        passed = .false.  ! should be an error?
      end if
    else

```

```

        passed = .true.
    end if
end subroutine subevt_expr_evaluate

```

### 15.3.4 Implementation for partonic events

This implementation contains the expressions that we can evaluate for the partonic process during integration.

```

⟨Subevt expr: public⟩≡
    public :: parton_expr_t

⟨Subevt expr: types⟩+≡
    type, extends (subevt_expr_t) :: parton_expr_t
        integer, dimension(:), allocatable :: i_beam
        integer, dimension(:), allocatable :: i_in
        integer, dimension(:), allocatable :: i_out
        logical :: has_scale = .false.
        logical :: has_fac_scale = .false.
        logical :: has_ren_scale = .false.
        logical :: has_weight = .false.
        type(eval_tree_t) :: scale
        type(eval_tree_t) :: fac_scale
        type(eval_tree_t) :: ren_scale
        type(eval_tree_t) :: weight
    contains
        ⟨Subevt expr: parton expr: TBP⟩
    end type parton_expr_t

```

Finalizer.

```

⟨Subevt expr: parton expr: TBP⟩≡
    procedure :: final => parton_expr_final

⟨Subevt expr: procedures⟩+≡
    subroutine parton_expr_final (object)
        class(parton_expr_t), intent(inout) :: object
        call object%base_final ()
        if (object%has_scale) then
            call eval_tree_final (object%scale)
        end if
        if (object%has_fac_scale) then
            call eval_tree_final (object%fac_scale)
        end if
        if (object%has_ren_scale) then
            call eval_tree_final (object%ren_scale)
        end if
        if (object%has_weight) then
            call eval_tree_final (object%weight)
        end if
    end subroutine parton_expr_final

```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subevt_t` base type. Here, it is unused.

```

<Subevt expr: parton expr: TBP>+≡
  procedure :: write => parton_expr_write

<Subevt expr: procedures>+≡
  subroutine parton_expr_write (object, unit, prefix)
    class(parton_expr_t), intent(in) :: object
    integer, intent(in), optional :: unit
    character(*), intent(in), optional :: prefix
    integer :: u
    u = output_unit (unit)
    call object%base_write (u)
    if (object%subevt_filled) then
      if (object%has_scale) then
        call write_separator (u)
        write (u, "(1x,A)") "Scale expression:"
        call write_separator (u)
        call eval_tree_write (object%scale, u)
      end if
      if (object%has_fac_scale) then
        call write_separator (u)
        write (u, "(1x,A)") "Factorization scale expression:"
        call write_separator (u)
        call eval_tree_write (object%fac_scale, u)
      end if
      if (object%has_ren_scale) then
        call write_separator (u)
        write (u, "(1x,A)") "Renormalization scale expression:"
        call write_separator (u)
        call eval_tree_write (object%ren_scale, u)
      end if
      if (object%has_weight) then
        call write_separator (u)
        write (u, "(1x,A)") "Weight expression:"
        call write_separator (u)
        call eval_tree_write (object%weight, u)
      end if
    end if
  end subroutine parton_expr_write

```

Define variables.

```

<Subevt expr: parton expr: TBP>+≡
  procedure :: setup_vars => parton_expr_setup_vars

<Subevt expr: procedures>+≡
  subroutine parton_expr_setup_vars (expr, sqrts)
    class(parton_expr_t), intent(inout), target :: expr
    real(default), intent(in) :: sqrts
    call expr%base_setup_vars (sqrts)
  end subroutine parton_expr_setup_vars

```

Compile the scale expressions. If a pointer is disassociated, there is no expression.

```

<Subevt expr: parton expr: TBP>+≡
  procedure :: setup_scales => parton_expr_setup_scales

<Subevt expr: procedures>+≡
  subroutine parton_expr_setup_scales &
    (expr, pn_scale, pn_fac_scale, pn_ren_scale)
    class(parton_expr_t), intent(inout), target :: expr
    type(parse_node_t), intent(in), pointer :: pn_scale
    type(parse_node_t), intent(in), pointer :: pn_fac_scale, pn_ren_scale
    if (associated (pn_scale)) then
      call eval_tree_init_expr (expr%scale, &
        pn_scale, expr%var_list, expr%subevt_t)
      expr%has_scale = .true.
    end if
    if (associated (pn_fac_scale)) then
      call eval_tree_init_expr (expr%fac_scale, &
        pn_fac_scale, expr%var_list, expr%subevt_t)
      expr%has_fac_scale = .true.
    end if
    if (associated (pn_ren_scale)) then
      call eval_tree_init_expr (expr%ren_scale, &
        pn_ren_scale, expr%var_list, expr%subevt_t)
      expr%has_ren_scale = .true.
    end if
  end subroutine parton_expr_setup_scales

```

Compile the weight expression.

```

<Subevt expr: parton expr: TBP>+≡
  procedure :: setup_weight => parton_expr_setup_weight

<Subevt expr: procedures>+≡
  subroutine parton_expr_setup_weight (expr, pn_weight)
    class(parton_expr_t), intent(inout), target :: expr
    type(parse_node_t), intent(in), pointer :: pn_weight
    if (associated (pn_weight)) then
      call eval_tree_init_expr (expr%weight, &
        pn_weight, expr%var_list, expr%subevt_t)
      expr%has_weight = .true.
    end if
  end subroutine parton_expr_setup_weight

```

Filling the partonic state consists of two parts. The first routine prepares the subevt without assigning momenta. It takes the particles from an `interaction_t`. It needs the indices and flavors for the beam, incoming, and outgoing particles.

We can assume that the particle content of the subevt does not change. Therefore, we set the event variables `n_in`, `n_out`, `n_tot` already in this initialization step.

```

<Subevt expr: parton expr: TBP>+≡
  procedure :: setup_subevt => parton_expr_setup_subevt

```

*<Subevt expr: procedures>+≡*

```

subroutine parton_expr_setup_subevt (expr, int, &
    i_beam, i_in, i_out, f_beam, f_in, f_out)
    class(parton_expr_t), intent(inout) :: expr
    type(interaction_t), intent(in), target :: int
    integer, dimension(:), intent(in) :: i_beam, i_in, i_out
    type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
    allocate (expr%i_beam (size (i_beam)))
    allocate (expr%i_in (size (i_in)))
    allocate (expr%i_out (size (i_out)))
    expr%i_beam = i_beam
    expr%i_in = i_in
    expr%i_out = i_out
    call interaction_to_subevt (int, &
        expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
    call subevt_set_pdg_beam      (expr%subevt_t, flavor_get_pdg (f_beam))
    call subevt_set_pdg_incoming (expr%subevt_t, flavor_get_pdg (f_in))
    call subevt_set_pdg_outgoing (expr%subevt_t, flavor_get_pdg (f_out))
    call subevt_set_p2_beam      (expr%subevt_t, flavor_get_mass (f_beam) ** 2)
    call subevt_set_p2_incoming (expr%subevt_t, flavor_get_mass (f_in) ** 2)
    call subevt_set_p2_outgoing (expr%subevt_t, flavor_get_mass (f_out) ** 2)
    expr%n_in = size (i_in)
    expr%n_out = size (i_out)
    expr%n_tot = expr%n_in + expr%n_out
end subroutine parton_expr_setup_subevt

```

The second part takes the momenta from the interaction object and thus completes the subevt. The partonic energy can then be computed.

*<Subevt expr: parton expr: TBP>+≡*

```

procedure :: fill_subevt => parton_expr_fill_subevt

```

*<Subevt expr: procedures>+≡*

```

subroutine parton_expr_fill_subevt (expr, int)
    class(parton_expr_t), intent(inout) :: expr
    type(interaction_t), intent(in), target :: int
    call interaction_momenta_to_subevt (int, &
        expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
    expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
    expr%subevt_filled = .true.
end subroutine parton_expr_fill_subevt

```

Evaluate, if the event passes the selection. For absent expressions we take default values.

*<Subevt expr: parton expr: TBP>+≡*

```

procedure :: evaluate => parton_expr_evaluate

```

*<Subevt expr: procedures>+≡*

```

subroutine parton_expr_evaluate &
    (expr, passed, scale, fac_scale, ren_scale, weight)
    class(parton_expr_t), intent(inout) :: expr
    logical, intent(out) :: passed
    real(default), intent(out) :: scale
    real(default), intent(out) :: fac_scale
    real(default), intent(out) :: ren_scale

```



```

real(default), intent(out) :: weight
call expr%base_evaluate (passed)
if (passed) then
  if (expr%has_scale) then
    call eval_tree_evaluate (expr%scale)
    if (eval_tree_result_is_known (expr%scale)) then
      scale = eval_tree_get_real (expr%scale)
    else
      scale = 0 ! should be an error?
    end if
  else
    scale = expr%sqrts_hat
  end if
  if (expr%has_fac_scale) then
    call eval_tree_evaluate (expr%fac_scale)
    if (eval_tree_result_is_known (expr%fac_scale)) then
      fac_scale = eval_tree_get_real (expr%fac_scale)
    else
      fac_scale = 0 ! should be an error?
    end if
  else
    fac_scale = scale
  end if
  if (expr%has_ren_scale) then
    call eval_tree_evaluate (expr%ren_scale)
    if (eval_tree_result_is_known (expr%ren_scale)) then
      ren_scale = eval_tree_get_real (expr%ren_scale)
    else
      ren_scale = 0 ! should be an error?
    end if
  else
    ren_scale = scale
  end if
  if (expr%has_weight) then
    call eval_tree_evaluate (expr%weight)
    if (eval_tree_result_is_known (expr%weight)) then
      weight = eval_tree_get_real (expr%weight)
    else
      weight = 0 ! should be an error?
    end if
  else
    weight = 1
  end if
end if
end subroutine parton_expr_evaluate

```

Return the beam/incoming parton indices.

*(Subevt expr: parton expr: TBP)+≡*

```

procedure :: get_beam_index => parton_expr_get_beam_index
procedure :: get_in_index => parton_expr_get_in_index

```

*(Subevt expr: procedures)+≡*

```

subroutine parton_expr_get_beam_index (expr, i_beam)
class(parton_expr_t), intent(in) :: expr

```

```

integer, dimension(:), intent(out) :: i_beam
i_beam = expr%i_beam
end subroutine parton_expr_get_beam_index

subroutine parton_expr_get_in_index (expr, i_in)
class(parton_expr_t), intent(in) :: expr
integer, dimension(:), intent(out) :: i_in
i_in = expr%i_in
end subroutine parton_expr_get_in_index

```

### 15.3.5 Implementation for full events

This implementation contains the expressions that we can evaluate for the full event.

```

<Subvt expr: public>+≡
public :: event_expr_t

<Subvt expr: types>+≡
type, extends (subvt_expr_t) :: event_expr_t
logical :: has_reweight = .false.
logical :: has_analysis = .false.
type(eval_tree_t) :: reweight
type(eval_tree_t) :: analysis
contains
<Subvt expr: event expr: TBP>
end type event_expr_t

```

Finalizer.

```

<Subvt expr: event expr: TBP>≡
procedure :: final => event_expr_final

<Subvt expr: procedures>+≡
subroutine event_expr_final (object)
class(event_expr_t), intent(inout) :: object
call object%base_final ()
if (object%has_reweight) then
call eval_tree_final (object%reweight)
end if
if (object%has_analysis) then
call eval_tree_final (object%analysis)
end if
end subroutine event_expr_final

```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subvt_t` base type. Here, it is unused.

```

<Subvt expr: event expr: TBP>+≡
procedure :: write => event_expr_write

```

```

<Subvt expr: procedures>+≡
  subroutine event_expr_write (object, unit, prefix)
    class(event_expr_t), intent(in) :: object
    integer, intent(in), optional :: unit
    character(*), intent(in), optional :: prefix
    integer :: u
    u = output_unit (unit)
    call object%base_write (u)
    if (object%subvt_filled) then
      if (object%has_reweight) then
        call write_separator (u)
        write (u, "(1x,A)") "Reweighting expression:"
        call write_separator (u)
        call eval_tree_write (object%reweight, u)
      end if
      if (object%has_analysis) then
        call write_separator (u)
        write (u, "(1x,A)") "Analysis expression:"
        call write_separator (u)
        call eval_tree_write (object%analysis, u)
      end if
    end if
  end subroutine event_expr_write

```

Define variables.

```

<Subvt expr: event expr: TBP>+≡
  procedure :: setup_vars => event_expr_setup_vars

<Subvt expr: procedures>+≡
  subroutine event_expr_setup_vars (expr, sqrts)
    class(event_expr_t), intent(inout), target :: expr
    real(default), intent(in) :: sqrts
    call expr%base_setup_vars (sqrts)
  end subroutine event_expr_setup_vars

```

Compile the analysis expression. If the pointer is disassociated, there is no expression.

```

<Subvt expr: event expr: TBP>+≡
  procedure :: setup_analysis => event_expr_setup_analysis

<Subvt expr: procedures>+≡
  subroutine event_expr_setup_analysis (expr, pn_analysis)
    class(event_expr_t), intent(inout), target :: expr
    type(parse_node_t), intent(in), pointer :: pn_analysis
    if (associated (pn_analysis)) then
      call eval_tree_init_lexpr (expr%analysis, &
        pn_analysis, expr%var_list, expr%subvt_t)
      expr%has_analysis = .true.
    end if
  end subroutine event_expr_setup_analysis

```

Compile the reweight expression.

```

<Subvt expr: event expr: TBP>+≡
  procedure :: setup_reweight => event_expr_setup_reweight

```

```

<Subevt expr: procedures>+≡
  subroutine event_expr_setup_reweight (expr, pn_reweight)
    class(event_expr_t), intent(inout), target :: expr
    type(parse_node_t), intent(in), pointer :: pn_reweight
    if (associated (pn_reweight)) then
      call eval_tree_init_expr (expr%reweight, &
        pn_reweight, expr%var_list, expr%subevt_t)
      expr%has_reweight = .true.
    end if
  end subroutine event_expr_setup_reweight

```

Fill the event expression: take the particle data and kinematics from a `particle_set` object.

We allow the particle content to change for each event. Therefore, we set the event variables each time.

```

<Subevt expr: event expr: TBP>+≡
  procedure :: fill_subevt => event_expr_fill_subevt

<Subevt expr: procedures>+≡
  subroutine event_expr_fill_subevt (expr, particle_set)
    class(event_expr_t), intent(inout) :: expr
    type(particle_set_t), intent(in) :: particle_set
    call particle_set_to_subevt (particle_set, expr%subevt_t)
    expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
    expr%n_in = particle_set_get_n_in (particle_set)
    expr%n_out = particle_set_get_n_out (particle_set)
    expr%n_tot = expr%n_in + expr%n_out
    expr%subevt_filled = .true.
  end subroutine event_expr_fill_subevt

```

Evaluate, if the event passes the selection. For absent expressions we take default values.

```

<Subevt expr: event expr: TBP>+≡
  procedure :: evaluate => event_expr_evaluate

<Subevt expr: procedures>+≡
  subroutine event_expr_evaluate (expr, passed, reweight, analysis_flag)
    class(event_expr_t), intent(inout) :: expr
    logical, intent(out) :: passed
    real(default), intent(out) :: reweight
    logical, intent(out) :: analysis_flag
    call expr%base_evaluate (passed)
    if (passed) then
      if (expr%has_reweight) then
        call eval_tree_evaluate (expr%reweight)
        if (eval_tree_result_is_known (expr%reweight)) then
          reweight = eval_tree_get_real (expr%reweight)
        else
          reweight = 0 ! should be an error?
        end if
      else
        reweight = 1
      end if
    end if
  end subroutine event_expr_evaluate

```

```

    if (expr%has_analysis) then
      call eval_tree_evaluate (expr%analysis)
      if (eval_tree_result_is_known (expr%analysis)) then
        analysis_flag = eval_tree_get_log (expr%analysis)
      else
        analysis_flag = .false.  ! should be an error?
      end if
    else
      analysis_flag = .true.
    end if
  end if
end subroutine event_expr_evaluate

```

### 15.3.6 Auxiliary stuff

Write a separator line.

```

<Subvt expr: procedures>+≡
  subroutine write_separator (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("-", 72)
  end subroutine write_separator

  subroutine write_separator_double (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("=", 72)
  end subroutine write_separator_double

```

### 15.3.7 Test

This is the master for calling self-test procedures.

```

<Subvt expr: public>+≡
  public :: subvt_expr_test

<Subvt expr: tests>≡
  subroutine subvt_expr_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <Subvt expr: execute tests>
  end subroutine subvt_expr_test

```

### Parton-event expressions

```

<Subvt expr: execute tests>≡
  call test (subvt_expr_1, "subvt_expr_1", &
    "parton-event expressions", &
    u, results)

```

*<Subevt expr: tests>+≡*

```

subroutine subevt_expr_1 (u)
  integer, intent(in) :: u
  type(string_t) :: expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(parse_tree_t) :: pt_cuts, pt_scale, pt_fac_scale, pt_ren_scale
  type(parse_tree_t) :: pt_weight
  type(parse_node_t), pointer :: pn_cuts, pn_scale, pn_fac_scale, pn_ren_scale
  type(parse_node_t), pointer :: pn_weight
  type(os_data_t) :: os_data
  type(model_t), pointer :: model => null ()
  type(parton_expr_t), target :: expr
  real(default) :: E, Ex, m
  type(vector4_t), dimension(6) :: p
  integer :: i, pdg
  logical :: passed
  real(default) :: scale, fac_scale, ren_scale, weight

  write (u, "(A)")  "* Test output: subevt_expr_1"
  write (u, "(A)")  "* Purpose: Set up a subevt and associated &
    &process-specific expressions"
  write (u, "(A)")

  call syntax_pexpr_init ()
  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)

  write (u, "(A)")  "* Expression texts"
  write (u, "(A)")

  expr_text = "all Pt > 100 [s]"
  write (u, "(A,A)")  "cuts = ", char (expr_text)
  call ifile_clear (ifile)
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_lexpr (pt_cuts, stream, .true.)
  call stream_final (stream)
  pn_cuts => parse_tree_get_root_ptr (pt_cuts)

  expr_text = "sqrts"
  write (u, "(A,A)")  "scale = ", char (expr_text)
  call ifile_clear (ifile)
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_expr (pt_scale, stream, .true.)
  call stream_final (stream)
  pn_scale => parse_tree_get_root_ptr (pt_scale)

  expr_text = "sqrts_hat"
  write (u, "(A,A)")  "fac_scale = ", char (expr_text)

```

```

call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
call stream_final (stream)
pn_fac_scale => parse_tree_get_root_ptr (pt_fac_scale)

expr_text = "100"
write (u, "(A,A)") "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)
pn_ren_scale => parse_tree_get_root_ptr (pt_ren_scale)

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)") "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)
pn_weight => parse_tree_get_root_ptr (pt_weight)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)") "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call expr%link_var_list (model_get_var_list_ptr (model))

call expr%setup_selection (pn_cuts)
call expr%setup_scales (pn_scale, pn_fac_scale, pn_ren_scale)
call expr%setup_weight (pn_weight)

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)") "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
m = 125._default
pdg = 25
p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)

```

```

p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)

call expr%reset ()
do i = 1, 2
  call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 3, 4
  call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 5, 6
  call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
end do
expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
expr%n_in = 2
expr%n_out = 2
expr%n_tot = 4
expr%subevt_filled = .true.

call expr%evaluate (passed, scale, fac_scale, ren_scale, weight)

write (u, "(A,L1)")      "Event has passed      = ", passed
write (u, "(A,ES12.5)")  "Scale                  = ", scale
write (u, "(A,ES12.5)")  "Factorization scale   = ", fac_scale
write (u, "(A,ES12.5)")  "Renormalization scale = ", ren_scale
write (u, "(A,ES12.5)")  "Weight                 = ", weight
write (u, "(A)")

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()
call expr%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: subevt_expr_1"

end subroutine subevt_expr_1

```

## Parton-event expressions

```

<Subevt expr: execute tests>+≡
  call test (subevt_expr_2, "subevt_expr_2", &
    "parton-event expressions", &
    u, results)
<Subevt expr: tests>+≡
  subroutine subevt_expr_2 (u)

```



```

integer, intent(in) :: u
type(string_t) :: expr_text
type(ifile_t) :: ifile
type(stream_t) :: stream
type(parse_tree_t) :: pt_selection
type(parse_tree_t) :: pt_reweight, pt_analysis
type(parse_node_t), pointer :: pn_selection
type(parse_node_t), pointer :: pn_reweight, pn_analysis
type(os_data_t) :: os_data
type(model_t), pointer :: model => null ()
type(event_expr_t), target :: expr
real(default) :: E, Ex, m
type(vector4_t), dimension(6) :: p
integer :: i, pdg
logical :: passed
real(default) :: reweight
logical :: analysis_flag

write (u, "(A)")  "* Test output: subevt_expr_2"
write (u, "(A)")  "* Purpose: Set up a subevt and associated &
    &process-specific expressions"
write (u, "(A)")

call syntax_pexpr_init ()
call syntax_model_file_init ()
call os_data_init (os_data)
call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), os_data, model)

write (u, "(A)")  "* Expression texts"
write (u, "(A)")

expr_text = "all Pt > 100 [s]"
write (u, "(A,A)") "selection = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_selection, stream, .true.)
call stream_final (stream)
pn_selection => parse_tree_get_root_ptr (pt_selection)

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)") "reweight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)
pn_reweight => parse_tree_get_root_ptr (pt_reweight)

expr_text = "true"
write (u, "(A,A)") "analysis = ", char (expr_text)
call ifile_clear (ifile)

```

```

call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)
pn_analysis => parse_tree_get_root_ptr (pt_analysis)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call expr%link_var_list (model_get_var_list_ptr (model))

call expr%setup_selection (pn_selection)
call expr%setup_analysis (pn_analysis)
call expr%setup_reweight (pn_reweight)

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
m = 125._default
pdg = 25
p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)
p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)

call expr%reset ()
do i = 1, 2
  call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 3, 4
  call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 5, 6
  call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
end do
expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
expr%n_in = 2
expr%n_out = 2
expr%n_tot = 4
expr%subevt_filled = .true.

```

```

call expr%evaluate (passed, reweight, analysis_flag)

write (u, "(A,L1)")      "Event has passed      = ", passed
write (u, "(A,ES12.5)")  "Reweighting factor   = ", reweight
write (u, "(A,L1)")      "Analysis flag        = ", analysis_flag
write (u, "(A)")

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call model_list_final ()
call expr%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: subevt_expr_2"

end subroutine subevt_expr_2

```

## 15.4 Parton states

A `parton_state_t` object contains the effective kinematics and dynamics of an elementary partonic interaction, with or without the beam/structure function state included. The type is abstract and has two distinct extensions. The `isolated_state_t` extension describes the isolated elementary interaction where the `int_eff` subobject contains the complex transition amplitude, exclusive in all quantum numbers. The particle content and kinematics describe the effective partonic state. The `connected_state_t` extension contains the partonic `subevt` and the expressions for cuts and scales which use it.

In the isolated state, the effective partonic interaction may either be identical to the hard interaction, in which case it is just a pointer to the latter. Or it may involve a rearrangement of partons, in which case we allocate it explicitly and flag this by `int_is_allocated`.

The `trace` evaluator contains the absolute square of the effective transition amplitude matrix, summed over final states. It is also summed over initial states, depending on the the beam setup allows. The result is used for integration.

The `matrix` evaluator is the counterpart of `trace` which is kept exclusive in all observable quantum numbers. The `flows` evaluator is furthermore exclusive in colors, but neglecting all color interference. The `matrix` and `flows` evaluators are filled only for sampling points that become part of physical events.

Note: It would be natural to make the evaluators allocatable. However, this causes memory corruption in gfortran 4.6.3. The extra `has_XXX` flags indicate whether evaluators are active, instead.

This module contains no unit tests. The tests are covered by the `processes` module below.

```

⟨parton_states.f90⟩≡
  ⟨File header⟩
  module parton_states

    ⟨Use kinds⟩
    ⟨Use strings⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use parser
    use lorentz !NODEP!
    use subevents
    use variables
    use expressions
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use evaluators

    use beams
    use sf_base
    use process_constants
    use prc_core
    use subevt_expr

    ⟨Standard module head⟩

    ⟨Parton states: public⟩

    ⟨Parton states: types⟩

    contains

    ⟨Parton states: procedures⟩

  end module parton_states

```

### 15.4.1 Abstract base type

The common part are the evaluators, one for the trace (summed over all quantum numbers), one for the transition matrix (summed only over unobservable quantum numbers), and one for the flow distribution (transition matrix without interferences, exclusive in color flow).

```

⟨Parton states: types⟩≡
  type, abstract :: parton_state_t
    logical :: has_trace = .false.
    logical :: has_matrix = .false.
    logical :: has_flows = .false.
    type(enumerator_t) :: trace

```

```

    type(evaluator_t) :: matrix
    type(evaluator_t) :: flows
contains
  ⟨Parton states: parton state: TBP⟩
end type parton_state_t

```

The `isolated_state_t` extension contains the `sf_chain_eff` object and the (hard) effective interaction `int_eff`, separately, both implemented as a pointer. The evaluators (trace, matrix, flows) apply to the hard interaction only.

If the effective interaction differs from the hard interaction, the pointer is allocated explicitly. Analogously for `sf_chain_eff`.

```

⟨Parton states: public⟩≡
  public :: isolated_state_t

⟨Parton states: types⟩+≡
  type, extends (parton_state_t) :: isolated_state_t
    logical :: sf_chain_is_allocated = .false.
    type(sf_chain_instance_t), pointer :: sf_chain_eff => null ()
    logical :: int_is_allocated = .false.
    type(interaction_t), pointer :: int_eff => null ()
contains
  ⟨Parton states: isolated state: TBP⟩
end type isolated_state_t

```

The `connected_state_t` extension contains all data that enable the evaluation of observables for the effective connected state. The evaluators connect the (effective) structure-function chain and hard interaction that were kept separate in the `isolated_state_t`.

The `flows_sf` evaluator is an extended copy of the structure-function

The `expr` subobject consists of the `subevt`, a simple event record, expressions for cuts etc. which refer to this record, and a `var_list` which contains event-specific variables, linked to the process variable list. Variables used within the expressions are looked up in `var_list`.

```

⟨Parton states: types⟩+≡
  public :: connected_state_t

⟨Parton states: types⟩+≡
  type, extends (parton_state_t) :: connected_state_t
    logical :: has_flows_sf = .false.
    type(evaluator_t) :: flows_sf
    type(parton_expr_t) :: expr
contains
  ⟨Parton states: connected state: TBP⟩
end type connected_state_t

```

Output: each evaluator is written only when it is active. The `sf_chain` is only written if it is explicitly allocated.

```

⟨Parton states: parton state: TBP⟩≡
  procedure :: write => parton_state_write

```

```

(Parton states: procedures)≡
subroutine parton_state_write (state, unit)
  class(parton_state_t), intent(in) :: state
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  select type (state)
  class is (isolated_state_t)
    if (state%sf_chain_is_allocated) then
      call write_separator (u)
      call state%sf_chain_eff%write (u)
    end if
    if (state%int_is_allocated) then
      call write_separator (u)
      write (u, "(1x,A)" &
        "Effective interaction:")
      call write_separator (u)
      call interaction_write (state%int_eff, u)
    end if
  class is (connected_state_t)
    if (state%has_flows_sf) then
      call write_separator (u)
      write (u, "(1x,A)" &
        "Evaluator (extension of the beam evaluator &
        &with color contractions):")
      call write_separator (u)
      call evaluator_write (state%flows_sf, u)
    end if
  end select
  if (state%has_trace) then
    call write_separator (u)
    write (u, "(1x,A)" &
      "Evaluator (trace of the squared transition matrix):")
    call write_separator (u)
    call evaluator_write (state%trace, u)
  end if
  if (state%has_matrix) then
    call write_separator (u)
    write (u, "(1x,A)" &
      "Evaluator (squared transition matrix):")
    call write_separator (u)
    call evaluator_write (state%matrix, u)
  end if
  if (state%has_flows) then
    call write_separator (u)
    write (u, "(1x,A)" &
      "Evaluator (squared color-flow matrix):")
    call write_separator (u)
    call evaluator_write (state%flows, u)
  end if
  select type (state)
  class is (connected_state_t)
    call write_separator (u)
    call state%expr%write (u)

```

```

    end select
end subroutine parton_state_write

```

Finalize interaction and evaluators, but only if allocated.

```

(Parton states: parton state: TBP)+≡
  procedure :: final => parton_state_final

(Parton states: procedures)+≡
  subroutine parton_state_final (state)
    class(parton_state_t), intent(inout) :: state
    if (state%has_flows) then
      call evaluator_final (state%flows)
      state%has_flows = .false.
    end if
    if (state%has_matrix) then
      call evaluator_final (state%matrix)
      state%has_matrix = .false.
    end if
    if (state%has_trace) then
      call evaluator_final (state%trace)
      state%has_trace = .false.
    end if
    select type (state)
    class is (connected_state_t)
      call state%expr%final ()
    class is (isolated_state_t)
      if (state%int_is_allocated) then
        call interaction_final (state%int_eff)
        deallocate (state%int_eff)
        state%int_is_allocated = .false.
      end if
      if (state%sf_chain_is_allocated) then
        call state%sf_chain_eff%final ()
      end if
    end select
  end subroutine parton_state_final

```

## 15.4.2 Common Initialization

Initialize the isolated parton state. In this version, the effective structure-function chain `sf_chain_eff` and the effective interaction `int_eff` both are trivial pointers to the seed structure-function chain and to the hard interaction, respectively.

```

(Parton states: isolated state: TBP)≡
  procedure :: init => isolated_state_init_pointers

(Parton states: procedures)+≡
  subroutine isolated_state_init_pointers (state, sf_chain, int)
    class(isolated_state_t), intent(out) :: state
    type(sf_chain_instance_t), intent(in), target :: sf_chain
    type(interaction_t), intent(in), target :: int
    state%sf_chain_eff => sf_chain

```

```

state%int_eff => int
end subroutine isolated_state_init_pointers

```

### 15.4.3 Evaluator initialization: isolated state

Create an evaluator for the trace of the squared transition matrix. The trace goes over all outgoing quantum numbers. Whether we trace over incoming quantum numbers other than color, depends on the given `qn_mask_in`.

Note: The option for explicitly computing the color factor table (`use_hi_cf` false; `nc` defined) is disabled. We take the color factor table from the hard matrix element data.

```

(Parton states: isolated state: TBP)+≡
  procedure :: setup_square_trace => isolated_state_setup_square_trace
(Parton states: procedures)+≡
  subroutine isolated_state_setup_square_trace (state, core, qn_mask_in, col)
    class(isolated_state_t), intent(inout), target :: state
    class(prc_core_t), intent(in) :: core
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    integer, dimension(:), intent(in) :: col
    !   logical, intent(in), optional :: use_hi_color_factors
    !   integer, intent(in), optional :: nc
    !   logical :: use_hi_cf
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    !   if (present (use_hi_color_factors)) then
    !     use_hi_cf = use_hi_color_factors
    !   else
    !     use_hi_cf = .false.
    !   end if
    associate (data => core%data)
      allocate (qn_mask (data%n_in + data%n_out))
      qn_mask(:data%n_in) = &
        new_quantum_numbers_mask (.false., .true., .false.) &
        .or. qn_mask_in
      qn_mask(data%n_in+1:) = &
        new_quantum_numbers_mask (.true., .true., .true.)
    !   if (use_hi_cf) then
    !     call evaluator_init_square (state%trace, &
    !       state%int_eff, qn_mask, &
    !       data%cf_index, data%color_factors, col)
    !   else
    !     call evaluator_init_square (hi%eval_trace, hi%int, qn_mask, nc=nc)
    !   end if
    end associate
    state%has_trace = .true.
  end subroutine isolated_state_setup_square_trace

```

Setup the evaluator for the transition matrix, exclusive in helicities where this is requested. In particular, we keep initial-state polarization unless the mask `qn_mask_in` says otherwise. For the final-state polarization, we look at the properties of the particle flavors individually.

```

(Parton states: isolated state: TBP)+≡

```



```

procedure :: setup_square_matrix => isolated_state_setup_square_matrix
<Parton states: procedures>+=
subroutine isolated_state_setup_square_matrix &
  (state, core, model, qn_mask_in, col)
  class(isolated_state_t), intent(inout), target :: state
  class(prc_core_t), intent(in) :: core
  type(model_t), intent(in), target :: model
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
  integer, dimension(:), intent(in) :: col
!   logical, intent(in), optional :: use_hi_color_factors
!   integer, intent(in), optional :: nc
!   logical :: use_hi_cf
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  logical :: helmask, helmask_hd
!   if (present (use_hi_color_factors)) then
!     use_hi_cf = use_hi_color_factors
!   else
!     use_hi_cf = .false.
!   end if
  associate (data => core%data)
    allocate (qn_mask (data%n_in + data%n_out))
    allocate (flv (data%n_flv))
    qn_mask(:data%n_in) = &
      new_quantum_numbers_mask (.false., .true., .false.) &
      .or. qn_mask_in
    do i = data%n_in + 1, data%n_in + data%n_out
      call flavor_init (flv, data%flv_state(i,:), model)
      if (.not. all (flavor_is_stable (flv))) then
        helmask = all (flavor_decays_isotropically (flv))
        helmask_hd = all (flavor_decays_diagonal (flv))
      else
        helmask = all (.not. flavor_is_polarized (flv))
        helmask_hd = .true.
      end if
      qn_mask(i) = new_quantum_numbers_mask (.false., .true., &
        helmask, mask_hd = helmask_hd)
    end do
!   if (use_hi_cf) then
!     call evaluator_init_square (state%matrix, &
!       state%int_eff, qn_mask, &
!       data%cf_index, data%color_factors, col)
!   else
!     call evaluator_init_square (hi%eval_sqme, hi%int, qn_mask, nc=nc)
!   end if
  end associate
  state%has_matrix = .true.
end subroutine isolated_state_setup_square_matrix

```

This procedure initializes the evaluator that computes the contributions to color flows, neglecting color interference. The incoming-particle mask can be used to sum over incoming flavor.

```

(Parton states: isolated state: TBP)+≡
  procedure :: setup_square_flows => isolated_state_setup_square_flows

(Parton states: procedures)+≡
  subroutine isolated_state_setup_square_flows (state, core, model, qn_mask_in)
    class(isolated_state_t), intent(inout), target :: state
    class(prc_core_t), intent(in) :: core
    type(model_t), intent(in), target :: model
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(flavor_t), dimension(:), allocatable :: flv
    integer :: i
    logical :: helmask, helmask_hd
    associate (data => core%data)
      allocate (qn_mask (data%n_in + data%n_out))
      allocate (flv (data%n_flv))
      qn_mask(:data%n_in) = &
        new_quantum_numbers_mask (.false., .false., .false.) &
        .or. qn_mask_in
      do i = data%n_in + 1, data%n_in + data%n_out
        call flavor_init (flv, data%flv_state(i,:), model)
        if (.not. all (flavor_is_stable (flv))) then
          helmask = all (flavor_decays_isotropically (flv))
          helmask_hd = all (flavor_decays_diagonal (flv))
        else
          helmask = all (.not. flavor_is_polarized (flv))
          helmask_hd = .true.
        end if
        qn_mask(i) = new_quantum_numbers_mask (.false., .false., &
          helmask, mask_hd = helmask_hd)
      end do
      call evaluator_init_square (state%flows, state%int_eff, qn_mask, &
        expand_color_flows = .true.)
    end associate
    state%has_flows = .true.
  end subroutine isolated_state_setup_square_flows

```

#### 15.4.4 Evaluator initialization: connected state

Setup a trace evaluator as a product of two evaluators (incoming state, effective interaction). In the result, all quantum numbers are summed over.

```

(Parton states: connected state: TBP)≡
  procedure :: setup_connected_trace => connected_state_setup_connected_trace

(Parton states: procedures)+≡
  subroutine connected_state_setup_connected_trace (state, isolated)
    class(connected_state_t), intent(inout), target :: state
    type(isolated_state_t), intent(in), target :: isolated
    type(quantum_numbers_mask_t) :: mask
    type(interaction_t), pointer :: sf_chain_int
    mask = new_quantum_numbers_mask (.true., .true., .true.)
    sf_chain_int => isolated%sf_chain_eff%get_out_int_ptr ()
    call evaluator_init_product &

```

```

        (state%trace, sf_chain_int, isolated%trace, mask, mask)
        state%has_trace = .true.
    end subroutine connected_state_setup_connected_trace

```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interaction). In the intermediate state, color and helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

```

(Parton states: connected state: TBP)+≡
    procedure :: setup_connected_matrix => connected_state_setup_connected_matrix

(Parton states: procedures)+≡
    subroutine connected_state_setup_connected_matrix (state, isolated)
        class(connected_state_t), intent(inout), target :: state
        type(isolated_state_t), intent(in), target :: isolated
        type(quantum_numbers_mask_t) :: mask
        type(interaction_t), pointer :: sf_chain_int
        mask = new_quantum_numbers_mask (.false., .true., .true.)
        sf_chain_int => isolated%sf_chain_eff%get_out_int_ptr ()
        call evaluator_init_product &
            (state%matrix, sf_chain_int, isolated%matrix, mask)
        state%has_matrix = .true.
    end subroutine connected_state_setup_connected_matrix

```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interaction). In the intermediate state, only helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

```

(Parton states: connected state: TBP)+≡
    procedure :: setup_connected_flows => connected_state_setup_connected_flows

(Parton states: procedures)+≡
    subroutine connected_state_setup_connected_flows (state, isolated)
        class(connected_state_t), intent(inout), target :: state
        type(isolated_state_t), intent(in), target :: isolated
        type(quantum_numbers_mask_t) :: mask
        type(interaction_t), pointer :: sf_chain_int
        mask = new_quantum_numbers_mask (.false., .false., .true.)
        sf_chain_int => isolated%sf_chain_eff%get_out_int_ptr ()
        call evaluator_init_color_contractions (state%flows_sf, sf_chain_int)
        call evaluator_init_product &
            (state%flows, state%flows_sf, isolated%flows, mask)
        state%has_flows_sf = .true.
        state%has_flows = .true.
    end subroutine connected_state_setup_connected_flows

```

### 15.4.5 Cuts and expressions

Set up the `subevt` that corresponds to the connected interaction. The index arrays refer to the interaction.

We assign the particles as follows: the beam particles are the first two (decay process: one) entries in the trace evaluator. The incoming partons are identified

by their link to the outgoing partons of the structure-function chain. The outgoing partons are those of the trace evaluator, which include radiated partons during the structure-function chain.

```

(Parton states: connected state: TBP)+≡
  procedure :: setup_subevt => connected_state_setup_subevt

(Parton states: procedures)+≡
  subroutine connected_state_setup_subevt (state, sf_chain, f_beam, f_in, f_out)
    class(connected_state_t), intent(inout), target :: state
    type(sf_chain_instance_t), intent(in), target :: sf_chain
    type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
    integer :: n_beam, n_in, n_out, n_vir, n_tot, i, j
    integer, dimension(:), allocatable :: i_beam, i_in, i_out
    integer :: sf_out_i
    type(interaction_t), pointer :: int, sf_int
    int => evaluator_get_int_ptr (state%trace)
    sf_int => sf_chain%get_out_int_ptr ()
    n_beam = size (f_beam)
    n_in = size (f_in)
    n_out = size (f_out)
    n_vir = interaction_get_n_vir (int)
    n_tot = interaction_get_n_tot (int)
    allocate (i_beam (n_beam), i_in (n_in), i_out (n_out))
    i_beam = [(i, i = 1, n_beam)]
    do j = 1, n_in
      sf_out_i = sf_chain%get_out_i (j)
      i_in(j) = interaction_find_link (int, sf_int, sf_out_i)
    end do
    i_out = [(i, i = n_vir + 1, n_tot)]
    call state%expr%setup_subevt (int, &
      i_beam, i_in, i_out, f_beam, f_in, f_out)
  end subroutine connected_state_setup_subevt

```

Initialize the variable list specific for this state/term. We insert event variables (`sqrts_hat`) and link the process variable list. The variable list acquires pointers to subobjects of `state`, which must therefore have a `target` attribute.

```

(Parton states: connected state: TBP)+≡
  procedure :: setup_var_list => connected_state_setup_var_list

(Parton states: procedures)+≡
  subroutine connected_state_setup_var_list (state, process_var_list, beam_data)
    class(connected_state_t), intent(inout), target :: state
    type(var_list_t), intent(in), target :: process_var_list
    type(beam_data_t), intent(in) :: beam_data
    call state%expr%setup_vars (beam_data_get_sqrts (beam_data))
    call state%expr%link_var_list (process_var_list)
  end subroutine connected_state_setup_var_list

```

Allocate the cut expression etc.

```

(Parton states: connected state: TBP)+≡
  procedure :: setup_expressions => connected_state_setup_expressions

```

```

<Parton states: procedures>+≡
  subroutine connected_state_setup_expressions (state, &
    pn_cuts, pn_scale, pn_fac_scale, pn_ren_scale, pn_weight)
    class(connected_state_t), intent(inout), target :: state
    type(parse_node_t), intent(in), pointer :: pn_cuts
    type(parse_node_t), intent(in), pointer :: pn_scale
    type(parse_node_t), intent(in), pointer :: pn_fac_scale
    type(parse_node_t), intent(in), pointer :: pn_ren_scale
    type(parse_node_t), intent(in), pointer :: pn_weight
    call state%expr%setup_selection (pn_cuts)
    call state%expr%setup_scales (pn_scale, pn_fac_scale, pn_ren_scale)
    call state%expr%setup_weight (pn_weight)
  end subroutine connected_state_setup_expressions

```

Reset the expression object: invalidate the subevt.

```

<Parton states: connected state: TBP>+≡
  procedure :: reset_expressions => connected_state_reset_expressions

<Parton states: procedures>+≡
  subroutine connected_state_reset_expressions (state)
    class(connected_state_t), intent(inout) :: state
    call state%expr%reset ()
  end subroutine connected_state_reset_expressions

```

## 15.4.6 Evaluation

Transfer momenta to the trace evaluator and fill the subevt with this effective kinematics, if applicable.

Note: we may want to apply a boost for the subevt.

```

<Parton states: parton state: TBP>+≡
  procedure :: receive_kinematics => parton_state_receive_kinematics

<Parton states: procedures>+≡
  subroutine parton_state_receive_kinematics (state)
    class(parton_state_t), intent(inout), target :: state
    type(interaction_t), pointer :: int
    if (state%has_trace) then
      call evaluator_receive_momenta (state%trace)
      select type (state)
      class is (connected_state_t)
        int => evaluator_get_int_ptr (state%trace)
        call state%expr%fill_subevt (int)
      end select
    end if
  end subroutine parton_state_receive_kinematics

```

Recover kinematics: We assume that the trace evaluator is filled with momenta. Send those momenta back to the sources, then fill the variables and subevent as above.

The incoming momenta of the connected state are not connected to the isolated state but to the beam interaction. Therefore, the incoming momenta

within the isolated state do not become defined, yet. Instead, we reconstruct the beam (and ISR) momentum configuration.

```

(Parton states: parton state: TBP)+≡
  procedure :: send_kinematics => parton_state_send_kinematics
(Parton states: procedures)+≡
  subroutine parton_state_send_kinematics (state)
    class(parton_state_t), intent(inout), target :: state
    type(interaction_t), pointer :: int
    real(default) :: s_hat
    if (state%has_trace) then
      call evaluator_send_momenta (state%trace)
      select type (state)
        class is (connected_state_t)
          int => evaluator_get_int_ptr (state%trace)
          call state%expr%fill_subevt (int)
        end select
      end if
    end subroutine parton_state_send_kinematics

```

Evaluate the expressions. The routine evaluates first the cut expression. If the event passes, it evaluates the other expressions. Where no expressions are defined, default values are inserted.

```

(Parton states: connected state: TBP)+≡
  procedure :: evaluate_expressions => connected_state_evaluate_expressions
(Parton states: procedures)+≡
  subroutine connected_state_evaluate_expressions (state, passed, &
    scale, fac_scale, ren_scale, weight)
    class(connected_state_t), intent(inout) :: state
    logical, intent(out) :: passed
    real(default), intent(out) :: scale, fac_scale, ren_scale, weight
    call state%expr%evaluate (passed, scale, fac_scale, ren_scale, weight)
  end subroutine connected_state_evaluate_expressions

```

Evaluate the structure-function chain, if it is allocated explicitly. The argument is the factorization scale.

If the chain is merely a pointer, the chain should already be evaluated at this point.

```

(Parton states: isolated state: TBP)+≡
  procedure :: evaluate_sf_chain => isolated_state_evaluate_sf_chain
(Parton states: procedures)+≡
  subroutine isolated_state_evaluate_sf_chain (state, fac_scale)
    class(isolated_state_t), intent(inout) :: state
    real(default), intent(in) :: fac_scale
    if (state%sf_chain_is_allocated) then
      call state%sf_chain_eff%evaluate (fac_scale)
    end if
  end subroutine isolated_state_evaluate_sf_chain

```

Evaluate the trace.

```

(Parton states: parton state: TBP)+≡
  procedure :: evaluate_trace => parton_state_evaluate_trace

```

```

(Parton states: procedures) +=
  subroutine parton_state_evaluate_trace (state)
    class(parton_state_t), intent(inout) :: state
    if (state%has_trace) then
      call evaluator_evaluate (state%trace)
    end if
  end subroutine parton_state_evaluate_trace

```

Evaluate the extra evaluators that we need for physical events.

```

(Parton states: parton state: TBP) +=
  procedure :: evaluate_event_data => parton_state_evaluate_event_data

(Parton states: procedures) +=
  subroutine parton_state_evaluate_event_data (state)
    class(parton_state_t), intent(inout) :: state
    select type (state)
    type is (connected_state_t)
      if (state%has_flows_sf) then
        call evaluator_receive_momenta (state%flows_sf)
        call evaluator_evaluate (state%flows_sf)
      end if
    end select
    if (state%has_matrix) then
      call evaluator_receive_momenta (state%matrix)
      call evaluator_evaluate (state%matrix)
    end if
    if (state%has_flows) then
      call evaluator_receive_momenta (state%flows)
      call evaluator_evaluate (state%flows)
    end if
  end subroutine parton_state_evaluate_event_data

```

## 15.4.7 Accessing the state

Three functions return a pointer to the event-relevant interactions.

```

(Parton states: parton state: TBP) +=
  procedure :: get_trace_int_ptr => parton_state_get_trace_int_ptr
  procedure :: get_matrix_int_ptr => parton_state_get_matrix_int_ptr
  procedure :: get_flows_int_ptr => parton_state_get_flows_int_ptr

(Parton states: procedures) +=
  function parton_state_get_trace_int_ptr (state) result (ptr)
    class(parton_state_t), intent(in), target :: state
    type(interaction_t), pointer :: ptr
    if (state%has_trace) then
      ptr => evaluator_get_int_ptr (state%trace)
    else
      ptr => null ()
    end if
  end function parton_state_get_trace_int_ptr

  function parton_state_get_matrix_int_ptr (state) result (ptr)
    class(parton_state_t), intent(in), target :: state

```

```

type(interaction_t), pointer :: ptr
if (state%has_matrix) then
    ptr => evaluator_get_int_ptr (state%matrix)
else
    ptr => null ()
end if
end function parton_state_get_matrix_int_ptr

function parton_state_get_flows_int_ptr (state) result (ptr)
class(parton_state_t), intent(in), target :: state
type(interaction_t), pointer :: ptr
if (state%has_flows) then
    ptr => evaluator_get_int_ptr (state%flows)
else
    ptr => null ()
end if
end function parton_state_get_flows_int_ptr

```

Return the indices of the beam particles and the outgoing particles within the trace (and thus, matrix and flows) evaluator, respectively.

```

(Parton states: connected state: TBP) +=
    procedure :: get_beam_index => connected_state_get_beam_index
    procedure :: get_in_index => connected_state_get_in_index

(Parton states: procedures) +=
    subroutine connected_state_get_beam_index (state, i_beam)
        class(connected_state_t), intent(in) :: state
        integer, dimension(:), intent(out) :: i_beam
        call state%expr%get_beam_index (i_beam)
    end subroutine connected_state_get_beam_index

    subroutine connected_state_get_in_index (state, i_in)
        class(connected_state_t), intent(in) :: state
        integer, dimension(:), intent(out) :: i_in
        call state%expr%get_in_index (i_in)
    end subroutine connected_state_get_in_index

```

### 15.4.8 Auxiliary stuff

Write a separator line.

```

(Parton states: procedures) +=
    subroutine write_separator (u)
        integer, intent(in) :: u
        write (u, "(A)") repeat ("-", 72)
    end subroutine write_separator

    subroutine write_separator_double (u)
        integer, intent(in) :: u
        write (u, "(A)") repeat ("=", 72)
    end subroutine write_separator_double

```



## 15.5 Complete Elementary Processes

This module combines hard interactions, phase space, and (for scatterings) structure functions and interfaces them to the integration module.

The process object implements the combination of a fixed beam and structure-function setup with a number of elementary processes. The latter are called process components. The process object represents an entity which is supposedly observable. It should be meaningful to talk about the cross section of a process.

The individual components of a process are, technically, processes themselves, but they may have unphysical cross sections which have to be added for a physical result. Process components may be exclusive tree-level elementary processes, dipole subtraction term, loop corrections, etc.

The beam and structure function setup is common to all process components. Thus, there is only one instance of this part.

The process may be a scattering process or a decay process. In the latter case, there are no structure functions, and the beam setup consists of a single particle. Otherwise, the two classes are treated on the same footing.

Once a sampling point has been chosen, a process determines a set of partons with a correlated density matrix of quantum numbers. In general, each sampling point will generate, for each process component, one or more distinct parton configurations. This is the **computed** state. The computed state is the subject of the multi-channel integration algorithm.

For NLO computations, it is necessary to project the computed states onto another set of parton configurations (e.g., by recombining certain pairs). This is the **observed** state. When computing partonic observables, the information is taken from the observed state.

For the purpose of event generation, we will later select one parton configuration from the observed state and collapse the correlated quantum state. This configuration is then dressed by applying parton shower, decays and hadronization. The decay chain, in particular, combines a scattering process with possible subsequent decay processes on the parton level, which are full-fledged process objects themselves.

```
<processes.f90>≡  
  <File header>  
  
  module processes  
  
    <Use kinds>  
    <Use strings>  
    use system_dependencies !NODEP!  
    use constants !NODEP!  
    <Use file utils>  
    use diagnostics !NODEP!  
    use unit_tests  
    use md5  
    ! use cputime  
    use os_interface  
  
    use ifiles  
    use lexers
```

```

use parser

use lorentz !NODEP!
use pdg_arrays
use subevents
use variables
use expressions
use models
use flavors
use helicities
use colors
use quantum_numbers
use state_matrices
use polarizations
use interactions
use evaluators
use particles
use beams
use sf_mappings
use sf_base
use process_constants
use phs_base
use rng_base
use mci_base

use prclib_interfaces
use prc_core_def
use process_libraries
use prc_test

use integration_results
use prc_core
use parton_states

⟨Standard module head⟩

⟨Processes: public⟩

⟨Processes: parameters⟩

⟨Processes: types⟩

⟨Processes: process part types⟩

⟨Processes: process type⟩

⟨Processes: variables⟩

⟨Processes: interfaces⟩

⟨Processes: test types⟩

contains

```

```
<Processes: procedures>
```

```
<Processes: tests>
```

```
end module processes
```

### 15.5.1 The Process Object

A process object is the workspace for creating process instances for event generation. After initialization, its contents are filled by integration passes which shape the integration grids and compute cross sections. Processes are set up initially from user-level configuration data. After calculating integrals and thus developing integration grid data, the program may use a process object or a copy of it for the purpose of generating events

The process object consists of several subobjects with their specific purposes. The corresponding types are defined below. (Technically, the subobject type definitions have to come before the process type definition, but with NOWEB magic we reverse this order here.)

The **type** determines whether we are considering a decay or a scattering process.

The **meta** object describes the process and its environment. All contents become fixed when the object is initialized.

The **config** object holds physical and technical configuration data that have been obtained during process initialization, and which are common to all process components.

The individual process components are configured in the **component** objects. These objects contain more configuration parameters and workspace, as needed for the specific process variant.

The **term** objects describe parton configurations which are technically used as phase-space points. Each process component may split into several terms with distinct kinematics and particle content. Furthermore, each term may project on a different physical state, e.g., by particle recombination. The **term** object provides the framework for this projection, for applying cuts, weight, and thus completing the process calculation.

The **beam\_config** object describes the incoming particles, either the decay mother or the scattering beams. It also contains the structure-function information.

The **mci\_entry** objects configure a MC input parameter set and integrator, each. The number of parameters depends on the process component and on the beam and structure-function setup.

```
<Processes: public>≡
```

```
public :: process_t
```

```
<Processes: process type>≡
```

```
type :: process_t
```

```
private
```

```
type(process_metadata_t) :: &
```

```
meta
```

```
type(process_config_data_t) :: &
```

```
config
```

```
type(process_component_t), dimension(:), allocatable :: &
```

```

        component
type(process_term_t), dimension(:), allocatable :: &
    term
type(process_beam_config_t) :: &
    beam_config
type(process_mci_entry_t), dimension(:), allocatable :: &
    mci_entry
contains
    <Processes: process: TBP>
end type process_t

```

This procedure is an important debugging and inspection tool; it is not used during normal operation. The process object is written to a file (identified by unit, which may also be standard output). Optional flags determine whether we show everything or just the interesting parts.

```

<Processes: process: TBP>≡
    procedure :: write => process_write

<Processes: procedures>≡
    subroutine process_write (process, unit, show_all, &
        show_var_list, &
        show_os_data, &
        show_model, show_expressions, &
        show_sfchain, &
        show_equivalences, show_history, show_histories, &
        show_forest, show_x, &
        show_subevt, show_evaluators)
class(process_t), intent(in) :: process
integer, intent(in), optional :: unit
logical, intent(in), optional :: show_all
logical, intent(in), optional :: show_var_list
logical, intent(in), optional :: show_os_data
logical, intent(in), optional :: show_model, show_expressions
logical, intent(in), optional :: show_sfchain
logical, intent(in), optional :: show_equivalences
logical, intent(in), optional :: show_history, show_histories
logical, intent(in), optional :: show_forest, show_x
logical, intent(in), optional :: show_subevt, show_evaluators
logical :: all
logical :: var_list
logical :: counters
logical :: os_data
logical :: model, expressions
logical :: sfchain
logical :: equivalences, history, histories
logical :: forest, x
logical :: subevt, evaluators
integer :: u, i
u = output_unit (unit)
if (present (show_all)) then
    all = show_all
else
    all = .false.
end if

```

```

var_list = .false.
counters = .true.
os_data = .false.
model = .false.
expressions = .false.
sfchain = .false.
equivalences = .false.
history = .false.
histories = .false.
forest = .false.
x = .false.
subevt = .false.
evaluators = .false.
if (present (show_var_list)) then
    all = .false.; var_list = show_var_list
end if
if (present (show_os_data)) then
    all = .false.; os_data = show_os_data
end if
if (present (show_model)) then
    all = .false.; model = show_model
end if
if (present (show_expressions)) then
    all = .false.; expressions = show_expressions
end if
if (present (show_sfchain)) then
    all = .false.; sfchain = show_sfchain
end if
if (present (show_equivalences)) then
    all = .false.; equivalences = show_equivalences
end if
if (present (show_history)) then
    all = .false.; history = show_history
end if
if (present (show_histories)) then
    all = .false.; histories = show_histories
end if
if (present (show_forest)) then
    all = .false.; forest = show_forest
end if
if (present (show_x)) then
    all = .false.; x = show_x
end if
if (present (show_subevt)) then
    all = .false.; subevt = show_subevt
end if
if (present (show_evaluators)) then
    all = .false.; x = show_evaluators
end if
if (all) then
    var_list = .true.
    model = .true.
    expressions = .true.
    sfchain = .true.

```

```

        equivalences = .true.
        history = .true.
        histories = .true.
        forest = .true.
        x = .true.
        subevt = .true.
        evaluators = .true.
    end if
    call write_separator_double (u)
    call process%meta%write (u, var_list)
    if (process%meta%type == PRC_UNKNOWN) then
        call write_separator_double (u)
        return
    else
        call write_separator (u)
    end if
    call process%config%write (u, counters, os_data, model, expressions)
    call write_separator_double (u)
    if (allocated (process%component)) then
        write (u, "(1x,A)") "Process component configuration:"
        call write_separator (u)
        do i = 1, size (process%component)
            call process%component(i)%write (u)
        end do
    else
        write (u, "(1x,A)") "Process component configuration: [undefined]"
    end if
    call write_separator_double (u)
    if (allocated (process%term)) then
        write (u, "(1x,A)") "Process term configuration:"
        call write_separator (u)
        do i = 1, size (process%term)
            call process%term(i)%write (u)
        end do
    else
        write (u, "(1x,A)") "Process term configuration: [undefined]"
    end if
    call write_separator_double (u)
    call process%beam_config%write (u)
    call write_separator_double (u)
    if (allocated (process%mci_entry)) then
        write (u, "(1x,A)") "Multi-channel integrator configurations:"
        call write_separator (u)
        do i = 1, size (process%mci_entry)
            write (u, "(1x,A,IO,A)") "MCI #", i, ":"
            call process%mci_entry(i)%write (u)
        end do
    end if
    call write_separator_double (u)
end subroutine process_write

```

Finalizer. Explicitly iterate over all subobjects that may contain allocated pointers.

```

<Processes: process: TBP>+=
  procedure :: final => process_final

<Processes: procedures>+=
  subroutine process_final (process)
    class(process_t), intent(inout) :: process
    integer :: i
    ! skip meta
    ! skip config
    if (allocated (process%component)) then
      do i = 1, size (process%component)
        call process%component(i)%final ()
      end do
    end if
    if (allocated (process%term)) then
      do i = 1, size (process%term)
        call process%term(i)%final ()
      end do
    end if
    call process%beam_config%final ()
    if (allocated (process%mc_i_entry)) then
      do i = 1, size (process%mc_i_entry)
        call process%mc_i_entry(i)%final ()
      end do
    end if
  end subroutine process_final

```

## Process setup

Initialize a process. We need a process library, the name of the process, and a run ID.

```

<Processes: process: TBP>+=
  procedure :: init => process_init

<Processes: procedures>+=
  subroutine process_init (process, proc_id, run_id, lib, os_data)
    class(process_t), intent(out) :: process
    type(string_t), intent(in) :: proc_id
    type(string_t), intent(in) :: run_id
    type(process_library_t), intent(in), target :: lib
    type(os_data_t), intent(in) :: os_data
    if (.not. lib%is_active ()) then
      call msg_bug ("Process init: inactive library not handled yet")
    end if
    if (.not. lib%contains (proc_id)) then
      call msg_fatal ("Process library doesn't contain process '" &
        // char (proc_id) // "'")
      return
    end if
    associate (meta => process%meta)
      call meta%init (proc_id, run_id, lib)
      call process%config%init (meta, os_data)
      allocate (process%component (meta%n_components))
    end associate
  end subroutine process_init

```

```

        end associate
    end subroutine process_init

```

Store a snapshot of a variable list.

```

<Processes: process: TBP>+≡
    procedure :: set_var_list => process_set_var_list

<Processes: procedures>+≡
    subroutine process_set_var_list (process, var_list)
        class(process_t), intent(inout) :: process
        type(var_list_t), intent(in) :: var_list
        call var_list_init_snapshot (process%meta%var_list, var_list)
    end subroutine process_set_var_list

```

Initialize the process components, one by one, using a template for the process core object. The template is taken only for allocating the correct type; the contents are set by extracting the process entry from the library.

```

<Processes: process: TBP>+≡
    procedure :: init_component => process_init_component

<Processes: procedures>+≡
    subroutine process_init_component &
        (process, index, core_template, mci_template, phs_config_template)
        class(process_t), intent(inout) :: process
        integer, intent(in) :: index
        class(prc_core_t), intent(in), allocatable :: core_template
        class(mci_t), intent(in), allocatable :: mci_template
        class(phs_config_t), intent(in), allocatable :: phs_config_template
        call process%component(index)%init (index, &
            process%meta, core_template, mci_template, phs_config_template)
    end subroutine process_init_component

```

Determine the process terms for each process component.

```

<Processes: process: TBP>+≡
    procedure :: setup_terms => process_setup_terms

<Processes: procedures>+≡
    subroutine process_setup_terms (process)
        class(process_t), intent(inout) :: process
        type(model_t), pointer :: model
        integer :: i, j, k
        integer, dimension(:), allocatable :: n_entry
        integer :: n_components, n_tot
        model => process%config%model
        n_components = process%meta%n_components
        allocate (n_entry (n_components))
        do i = 1, n_components
            n_entry(i) = process%component(i)%core%get_n_terms ()
        end do
        n_tot = sum (n_entry)
        allocate (process%term (n_tot))
        k = 0
        do i = 1, n_components

```



```

        associate (component => process%component(i))
        associate (core => component%core)
        allocate (component%i_term (n_entry(i)))
        do j = 1, n_entry(i)
            component%i_term(j) = k + j
            call process%term(k+j)%init (k+j, i, j, core, model)
        end do
        end associate
    end associate
    k = k + n_entry(i)
end do
process%config%n_terms = n_tot
end subroutine process_setup_terms

```

Initialize the beam setup. This is the trivial version where the incoming state of the matrix element coincides with the initial state of the process. For a scattering process, we need the c.m. energy, all other variables are set to their default values (no polarization, lab frame and c.m. frame coincide, etc.)

We assume that all components consistently describe a scattering process, i.e., two incoming particles.

Note: The current layout of the `beam_data_t` record requires that the flavor for each beam is unique. For processes with multiple flavors in the initial state, one has to set up beams explicitly. This restriction could be removed by extending the code in the `beams` module.

```

(Processes: process: TBP)+≡
    generic :: setup_beams => setup_beams_sqrts
    procedure :: setup_beams_sqrts => process_setup_beams_sqrts
(Processes: procedures)+≡
    subroutine process_setup_beams_sqrts (process, sqrts)
        class(process_t), intent(inout) :: process
        real(default), intent(in) :: sqrts
        integer, dimension(:,:), allocatable :: flv_state_in
        type(flavor_t), dimension(2) :: flv_in
        integer :: i
        allocate (flv_state_in (2, process%meta%n_components))
        do i = 1, process%meta%n_components
            flv_state_in (:,i) = process%component(i)%get_flv_state_in ()
        end do
        if (all (flv_state_in(1,:) == flv_state_in(1,1)) .and. &
            all (flv_state_in(2,:) == flv_state_in(2,1))) then
            call flavor_init (flv_in, flv_state_in(:,1), process%config%model)
            call process%beam_config%init_scattering (flv_in, sqrts)
        else
            call msg_fatal ("Setting up process '" &
                // char (process%meta%id) // "': inconsistent initial state")
        end if
    end subroutine process_setup_beams_sqrts

```

Initialize from a complete beam setup.

```

(Processes: process: TBP)+≡
    generic :: setup_beams => setup_beams_beam_data
    procedure :: setup_beams_beam_data => process_setup_beams_beam_data

```

```

<Processes: procedures>+≡
  subroutine process_setup_beams_beam_data (process, beam_data)
    class(process_t), intent(inout) :: process
    type(beam_data_t), intent(in) :: beam_data
    call process%beam_config%init_beam_data (beam_data)
  end subroutine process_setup_beams_beam_data

```

Notify the user about beam setup.

```

<Processes: process: TBP>+≡
  procedure :: beams_startup_message => process_beams_startup_message

<Processes: procedures>+≡
  subroutine process_beams_startup_message (process, unit)
    class(process_t), intent(in) :: process
    integer, intent(in), optional :: unit
    call process%beam_config%startup_message (unit)
  end subroutine process_beams_startup_message

```

We complete the kinematics configuration after the beam setup, but before we configure the chain of structure functions. The reason is that we need the total energy `sqrts` for the kinematics, but the structure-function setup requires the number of channels, which depends on the kinematics configuration. For instance, the kinematics module may return the need for parameterizing an s-channel resonance.

```

<Processes: process: TBP>+≡
  procedure :: configure_phs => process_configure_phs

<Processes: procedures>+≡
  subroutine process_configure_phs (process, rebuild)
    class(process_t), intent(inout) :: process
    logical, intent(in), optional :: rebuild
    real(default) :: sqrts
    integer :: i
    sqrts = process%get_sqrts ()
    do i = 1, process%meta%n_components
      call process%component(i)%configure_phs &
        (sqrts, process%beam_config, rebuild)
    end do
  end subroutine process_configure_phs

```

Insert the structure-function configuration data. First allocate the storage, then insert data one by one. The third procedure declares a mapping (of the MC input parameters) for a specific channel and structure-function combination.

We take the number of channels from the corresponding entry in the `config_data` section.

Otherwise, these are simple wrapper routines. The extra level in the call tree may allow for simple addressing of multiple concurrent beam configurations, not implemented currently.

If we do not want structure functions, we simply do not call those procedures.

```

<Processes: process: TBP>+≡
  procedure :: init_sf_chain => process_init_sf_chain
  generic :: set_sf_channel => set_sf_channel_single

```

```

procedure :: set_sf_channel_single => process_set_sf_channel
generic :: set_sf_channel => set_sf_channel_array
procedure :: set_sf_channel_array => process_set_sf_channel_array

<Processes: procedures>+≡
subroutine process_init_sf_chain (process, sf_config)
  class(process_t), intent(inout) :: process
  type(sf_config_t), dimension(:), intent(in) :: sf_config
  call process%beam_config%init_sf_chain (sf_config)
end subroutine process_init_sf_chain

subroutine process_set_sf_channel (process, c, sf_channel)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: c
  type(sf_channel_t), intent(in) :: sf_channel
  call process%beam_config%set_sf_channel (c, sf_channel)
end subroutine process_set_sf_channel

subroutine process_set_sf_channel_array (process, sf_channel)
  class(process_t), intent(inout) :: process
  type(sf_channel_t), dimension(:), intent(in) :: sf_channel
  integer :: c
  do c = 1, size (sf_channel)
    call process%beam_config%set_sf_channel (c, sf_channel(c))
  end do
end subroutine process_set_sf_channel_array

```

Notify about the structure-function setup.

```

<Processes: process: TBP>+≡
  procedure :: sf_startup_message => process_sf_startup_message

<Processes: procedures>+≡
subroutine process_sf_startup_message (process, unit)
  class(process_t), intent(in) :: process
  integer, intent(in), optional :: unit
  call process%beam_config%sf_startup_message (unit)
end subroutine process_sf_startup_message

```

As soon as both the kinematics configuration and the structure-function setup are complete, we match parameterizations (channels) for both. The matching entries are (re)set in the component phase-space configuration, while the structure-function configuration is left intact.

```

<Processes: process: TBP>+≡
  procedure :: match_channels => process_match_channels

<Processes: procedures>+≡
subroutine process_match_channels (process)
  class(process_t), intent(inout) :: process
  integer :: i
  do i = 1, process%meta%n_components
    call process%component(i)%match_channels (process%beam_config)
  end do
end subroutine process_match_channels

```

Determine the MC parameter set structure and the MCI configuration for each process component. We need data from the structure-function and phase-space setup, so those should be complete before this is called.

```

(Processes: process: TBP)+≡
  procedure :: setup_mci => process_setup_mci

(Processes: procedures)+≡
  subroutine process_setup_mci (process)
    class(process_t), intent(inout) :: process
    integer :: n_mci, i_mci
    integer :: i
    n_mci = 0
    do i = 1, process%meta%n_components
      if (process%component(i)%core%needs_mcset ()) then
        n_mci = n_mci + 1
        process%component(i)%i_mci = n_mci
      end if
    end do
    process%config%n_mci = n_mci
    allocate (process%mci_entry (n_mci))
    i_mci = 0
    do i = 1, process%meta%n_components
      if (process%component(i)%core%needs_mcset ()) then
        associate (component => process%component(i))
          i_mci = i_mci + 1
          associate (mci_entry => process%mci_entry(i_mci))
            call mci_entry%init (process%meta%type, &
              i_mci, i, component, process%beam_config)
          end associate
        end associate
      end if
    end do
  end subroutine process_setup_mci

```

Import a random-number generator state into a MCI entry component.

```

(Processes: process: TBP)+≡
  procedure :: import_rng => process_import_rng

(Processes: procedures)+≡
  subroutine process_import_rng (process, i_mci, rng)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    class(rng_t), intent(inout), allocatable :: rng
    call process%mci_entry(i_mci)%mci%import_rng (rng)
  end subroutine process_import_rng

```

Set cuts. This is a parse node, namely the right-hand side of the cut assignment. When creating an instance, we compile this into an evaluation tree. The parse node may be null.

```

(Processes: process: TBP)+≡
  procedure :: set_cuts => process_set_cuts

```

```

<Processes: procedures>+≡
  subroutine process_set_cuts (process, pn_cuts)
    class(process_t), intent(inout) :: process
    type(parse_node_t), intent(in), pointer :: pn_cuts
    process%config%pn_cuts => pn_cuts
  end subroutine process_set_cuts

```

Analogously for the other expressions.

```

<Processes: process: TBP>+≡
  procedure :: set_scale => process_set_scale
  procedure :: set_fac_scale => process_set_fac_scale
  procedure :: set_ren_scale => process_set_ren_scale
  procedure :: set_weight => process_set_weight

<Processes: procedures>+≡
  subroutine process_set_scale (process, pn_scale)
    class(process_t), intent(inout) :: process
    type(parse_node_t), intent(in), pointer :: pn_scale
    process%config%pn_scale => pn_scale
  end subroutine process_set_scale

  subroutine process_set_fac_scale (process, pn_fac_scale)
    class(process_t), intent(inout) :: process
    type(parse_node_t), intent(in), pointer :: pn_fac_scale
    process%config%pn_fac_scale => pn_fac_scale
  end subroutine process_set_fac_scale

  subroutine process_set_ren_scale (process, pn_ren_scale)
    class(process_t), intent(inout) :: process
    type(parse_node_t), intent(in), pointer :: pn_ren_scale
    process%config%pn_ren_scale => pn_ren_scale
  end subroutine process_set_ren_scale

  subroutine process_set_weight (process, pn_weight)
    class(process_t), intent(inout) :: process
    type(parse_node_t), intent(in), pointer :: pn_weight
    process%config%pn_weight => pn_weight
  end subroutine process_set_weight

```

## MD5 sum

The MD5 sum of the process object should reflect the state completely, including integration results. It is used for checking the integrity of event files. This global checksum includes checksums for the various parts. In particular, the MCI object receives a checksum that includes the configuration of all configuration parts relevant for an individual integration. This checksum is used for checking the integrity of integration grids.

We do not need MD5 sums for the process terms, since these are generated from the component definitions.

```

<Processes: process: TBP>+≡
  procedure :: compute_md5sum => process_compute_md5sum

```

```

<Processes: procedures>+=
  subroutine process_compute_md5sum (process)
    class(process_t), intent(inout) :: process
    integer :: i
    call process%config%compute_md5sum ()
    do i = 1, process%config%n_components
      call process%component(i)%compute_md5sum ()
    end do
    call process%beam_config%compute_md5sum ()
    do i = 1, process%config%n_mci
      call process%mci_entry(i)%compute_md5sum &
        (process%config, process%component, process%beam_config)
    end do
  end subroutine process_compute_md5sum

```

## Integration and event generation

Integrate the process, using a previously initialized process instance. We select one of the available MCI integrators by its index `i_mci` and thus integrate over (structure functions and) phase space for the associated (group of) process component(s).

The finalizer should be called after all integration passes have been completed. It will, for instance, write a summary of the integration results.

```

<Processes: process: TBP>+=
  procedure :: integrate => process_integrate
  procedure :: final_integration => process_final_integration

<Processes: procedures>+=
  subroutine process_integrate (process, instance, i_mci, n_it, n_calls, &
    adapt_grids, adapt_weights)
    class(process_t), intent(inout) :: process
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
    call process%mci_entry(i_mci)%integrate (instance, n_it, n_calls, &
      adapt_grids, adapt_weights)
  end subroutine process_integrate

  subroutine process_final_integration (process, i_mci)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%final_integration ()
  end subroutine process_final_integration

```

Initialize and finalize event generation for the specified MCI entry.

```

<Processes: process: TBP>+=
  procedure :: init_simulation => process_init_simulation
  procedure :: final_simulation => process_final_simulation

```

```

(Processes: procedures) +=
  subroutine process_init_simulation (process, instance, i_mci)
    class(process_t), intent(inout) :: process
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%init_simulation (instance)
  end subroutine process_init_simulation

  subroutine process_final_simulation (process, instance, i_mci)
    class(process_t), intent(in) :: process
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%final_simulation (instance)
  end subroutine process_final_simulation

```

Generate a weighted event. We select one of the available MCI integrators by its index *i\_mci* and thus generate an event for the associated (group of) process component(s). The arguments exactly correspond to the initializer and finalizer above.

The resulting event is stored in the *process\_instance* object, which also holds the workspace of the integrator.

Note: The *process* object is declared *intent(inout)* because it contains the random-number state, which changes for each event. Otherwise, all volatile data are inside the *instance* object.

```

(Processes: process: TBP) +=
  procedure :: generate_weighted_event => process_generate_weighted_event
  procedure :: generate_unweighted_event => process_generate_unweighted_event

(Processes: procedures) +=
  subroutine process_generate_weighted_event (process, instance, i_mci)
    class(process_t), intent(inout) :: process
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%generate_weighted_event (instance)
  end subroutine process_generate_weighted_event

  subroutine process_generate_unweighted_event (process, instance, i_mci)
    class(process_t), intent(inout) :: process
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%generate_unweighted_event (instance)
  end subroutine process_generate_unweighted_event

```

This replaces the event generation methods for the situation that the process instance object has been filled by other means (i.e., reading and/or recalculating its contents). We just have to fill in missing MCI data, especially the event weight.

```

(Processes: process: TBP) +=
  procedure :: recover_event => process_recover_event

(Processes: procedures) +=
  subroutine process_recover_event (process, instance, i_term)
    class(process_t), intent(inout) :: process

```

```

type(process_instance_t), intent(inout) :: instance
integer, intent(in) :: i_term
call process%mci_entry(instance%i_mci)%recover_event (instance, i_term)
end subroutine process_recover_event

```

## Retrieve process data

Return the current integral and error obtained by the integrator *i\_mci*.

```

(Processes: process: TBP) +=
  generic :: get_integral => get_integral_tot, get_integral_mci
  generic :: get_error => get_error_tot, get_error_mci
  procedure :: get_integral_tot => process_get_integral_tot
  procedure :: get_integral_mci => process_get_integral_mci
  procedure :: get_error_tot => process_get_error_tot
  procedure :: get_error_mci => process_get_error_mci

(Processes: procedures) +=
  function process_get_integral_mci (process, i_mci) result (integral)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_mci
    real(default) :: integral
    integral = process%mci_entry(i_mci)%get_integral ()
  end function process_get_integral_mci

  function process_get_error_mci (process, i_mci) result (error)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_mci
    real(default) :: error
    error = process%mci_entry(i_mci)%get_error ()
  end function process_get_error_mci

  function process_get_integral_tot (process) result (integral)
    class(process_t), intent(in) :: process
    real(default) :: integral
    integer :: i
    integral = 0
    do i = 1, size (process%mci_entry)
      integral = integral + process%mci_entry(i)%get_integral ()
    end do
  end function process_get_integral_tot

  function process_get_error_tot (process) result (error)
    class(process_t), intent(in) :: process
    real(default) :: error
    real(default) :: variance
    integer :: i
    variance = 0
    do i = 1, size (process%mci_entry)
      variance = variance + process%mci_entry(i)%get_error () ** 2
    end do
    error = sqrt (variance)
  end function process_get_error_tot

```



Return the MD5 sums that summarize the process component definitions. These values should be independent of parameters, beam details, expressions, etc. They can be used for checking the integrity of a process when reusing an old event file.

```

<Processes: process: TBP>+≡
    procedure :: get_md5sum_prc => process_get_md5sum_prc
<Processes: procedures>+≡
    function process_get_md5sum_prc (process, i_component) result (md5sum)
        class(process_t), intent(in) :: process
        integer, intent(in) :: i_component
        character(32) :: md5sum
        md5sum = process%component(i_component)%config%get_md5sum ()
    end function process_get_md5sum_prc

```

Return the MD5 sums that summarize the state of the MCI integrators. These values should encode all process data, integration and phase space configuration, etc., and the integration results. They can thus be used for checking the integrity of an event-generation setup when reusing an old event file.

```

<Processes: process: TBP>+≡
    procedure :: get_md5sum_mci => process_get_md5sum_mci
<Processes: procedures>+≡
    function process_get_md5sum_mci (process, i_mci) result (md5sum)
        class(process_t), intent(in) :: process
        integer, intent(in) :: i_mci
        character(32) :: md5sum
        md5sum = process%mci_entry(i_mci)%get_md5sum ()
    end function process_get_md5sum_mci

```

## 15.5.2 Metadata

This information describes the process and its environment. It is fixed upon initialization.

The `id` string is the name of the process object, as given by the user. The matrix element generator will use this string for naming Fortran procedures and types, so it should qualify as a Fortran name.

The `run_id` string distinguishes among several runs for the same process. It identifies process instances with respect to adapted integration grids and similar run-specific data. The run ID is kept when copying processes for creating instances, however, so it does not distinguish event samples.

The `var_list` is a snapshot of the variable list, taken at the point where the process was initialized.

The `lib` pointer accesses the process library where the process definition and the process driver are located.

The `lib_index` is the index of entry in the process library that corresponds to the current process.

The `component_id` array identifies the individual process components.

The `component_description` is an array of human-readable strings that characterize the process components, for instance `a`, `b => c`, `d`.

```

<Processes: process part types>≡

```

```

type :: process_metadata_t
  private
  integer :: type = PRC_UNKNOWN
  type(string_t) :: id
  type(string_t) :: run_id
  type(var_list_t) :: var_list
  type(process_library_t), pointer :: lib => null ()
  integer :: lib_index = 0
  integer :: n_components = 0
  type(string_t), dimension(:), allocatable :: component_id
  type(string_t), dimension(:), allocatable :: component_description
contains
  <Processes: process metadata: TBP>
end type process_metadata_t

```

Output: ID and run ID. We write the variable list only upon request.

```

<Processes: process metadata: TBP>≡
  procedure :: write => process_metadata_write

<Processes: procedures>+≡
  subroutine process_metadata_write (meta, u, var_list)
    class(process_metadata_t), intent(in) :: meta
    integer, intent(in) :: u
    logical, intent(in) :: var_list
    integer :: i
    select case (meta%type)
    case (PRC_UNKNOWN)
      write (u, "(1x,A)") "Process [undefined]"
      return
    case (PRC_DECAY)
      write (u, "(1x,A)", advance="no") "Process [decay]:"
    case (PRC_SCATTERING)
      write (u, "(1x,A)", advance="no") "Process [scattering]:"
    case default
      call msg_bug ("process_write: undefined process type")
    end select
    write (u, "(1x,A,A,A)") "'", char (meta%id), "'"
    write (u, "(3x,A,A,A)") "Run ID          = '", char (meta%run_id), "'"
    if (associated (meta%lib)) then
      write (u, "(3x,A,A,A)") "Library name = '", &
        char (meta%lib%get_name ()), "'"
    else
      write (u, "(3x,A)") "Library name = [not associated]"
    end if
    write (u, "(3x,A,I0)") "Process index = ", meta%lib_index
    if (allocated (meta%component_id)) then
      write (u, "(3x,A)") "Process components:"
      do i = 1, size (meta%component_id)
        write (u, "(5x,I0,9A)") i, ": '", &
          char (meta%component_id (i)), "' : ", &
          char (meta%component_description (i))
      end do
    end if
    call write_separator (u)
  end subroutine

```

```

    if (var_list) then
        write (u, "(1x,A)") "Variable list:"
        call write_separator (u)
        call var_list_write (meta%var_list, u)
    else
        write (u, "(1x,A)") "Variable list: [not shown]"
    end if
end subroutine process_metadata_write

```

Initialize.

*(Processes: process metadata: TBP)+≡*

```

    procedure :: init => process_metadata_init

```

*(Processes: procedures)+≡*

```

    subroutine process_metadata_init (meta, id, run_id, lib)
        class(process_metadata_t), intent(out) :: meta
        type(string_t), intent(in) :: id
        type(string_t), intent(in) :: run_id
        type(process_library_t), intent(in), target :: lib
        select case (lib%get_n_in (id))
            case (1); meta%type = PRC_DECAY
            case (2); meta%type = PRC_SCATTERING
            case default
                call msg_bug ("Process '" // char (id) // "': impossible n_in")
            end select
        meta%id = id
        meta%run_id = run_id
        meta%lib => lib
        meta%lib_index = lib%get_entry_index (id)
        call lib%get_component_list (id, meta%component_id)
        meta%n_components = size (meta%component_id)
        call lib%get_component_description_list (id, meta%component_description)
    end subroutine process_metadata_init

```

### 15.5.3 Generic Configuration Data

This information concerns physical and technical properties of the process. It is fixed upon initialization, using data from the process specification and the variable list.

The number **n\_in** is the number of incoming beam particles, simultaneously the number of incoming partons, 1 for a decay and 2 for a scattering process. (The number of outgoing partons may depend on the process component.)

The number **n\_components** is the number of components that constitute the current process.

The number **n\_terms** is the number of distinct contributions to the scattering matrix that constitute the current process. Each component may generate several terms.

The number **n\_mci** is the number of independent MC integration configurations that this process uses. Distinct process components that share a MCI configuration may be combined pointwise. (Nevertheless, a given MC variable set may correspond to several “nearby” kinematical configurations.) This is also

the number of distinct sampling-function results that this process can generate. Process components that use distinct variable sets are added only once after an integration pass has completed.

The `model` pointer identifies the physics model and its parameters. This is a pointer to an external object.

The `qcd` object contains all QCD-related data that may influence the process.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions for evaluating cuts and scales. The workspaces for evaluating those expressions are set up in the `effective_state` subobjects. Note that these are really pointers, so the actual nodes are not stored inside the process object.

The `md5sum` is taken and used to verify the process configuration when re-reading data from file.

```

<Processes: process part types>+=
  type :: process_config_data_t
    private
      integer :: n_in = 0
      integer :: n_components = 0
      integer :: n_terms = 0
      integer :: n_mci = 0
      type(os_data_t) :: os_data
      type(string_t) :: model_name
      type(model_t), pointer :: model => null ()
      type(parse_node_t), pointer :: pn_cuts => null ()
      type(parse_node_t), pointer :: pn_scale => null ()
      type(parse_node_t), pointer :: pn_fac_scale => null ()
      type(parse_node_t), pointer :: pn_ren_scale => null ()
      type(parse_node_t), pointer :: pn_weight => null ()
      character(32) :: md5sum = ""
    contains
      <Processes: process config data: TBP>
    end type process_config_data_t

```

Here, we may compress the expressions for cuts etc.

```

<Processes: process config data: TBP>=
  procedure :: write => process_config_data_write

<Processes: procedures>+=
  subroutine process_config_data_write (config, u, &
    counters, os_data, model, expressions)
    class(process_config_data_t), intent(in) :: config
    integer, intent(in) :: u
    logical, intent(in) :: counters
    logical, intent(in) :: os_data
    logical, intent(in) :: model
    logical, intent(in) :: expressions
    write (u, "(1x,A)") "Configuration data:"
    if (counters) then
      write (u, "(3x,A,I0)") "Number of incoming particles = ", &
        config%n_in
      write (u, "(3x,A,I0)") "Number of process components = ", &
        config%n_components
      write (u, "(3x,A,I0)") "Number of process terms      = ", &

```

```

        config%n_terms
        write (u, "(3x,A,I0)") "Number of MCI configurations = ", &
            config%n_mci
    end if
    if (os_data) then
        call os_data_write (config%os_data, u)
    end if
    if (associated (config%model)) then
        write (u, "(3x,A,A)") "Model = ", char (config%model_name)
        if (model) then
            call write_separator (u)
            call config%model%write (u)
            call write_separator (u)
        end if
    else
        write (u, "(3x,A,A,A)") "Model = ", char (config%model_name), &
            " [not associated]"
    end if
    call write_separator (u)
    if (expressions) then
        if (associated (config%pn_cuts)) then
            call write_separator (u)
            write (u, "(3x,A)") "Cut expression:"
            call config%pn_cuts%write (u)
        end if
        if (associated (config%pn_scale)) then
            call write_separator (u)
            write (u, "(3x,A)") "Scale expression:"
            call config%pn_scale%write (u)
        end if
        if (associated (config%pn_fac_scale)) then
            call write_separator (u)
            write (u, "(3x,A)") "Factorization scale expression:"
            call config%pn_fac_scale%write (u)
        end if
        if (associated (config%pn_ren_scale)) then
            call write_separator (u)
            write (u, "(3x,A)") "Renormalization scale expression:"
            call config%pn_ren_scale%write (u)
        end if
        if (associated (config%pn_weight)) then
            call write_separator (u)
            write (u, "(3x,A)") "Weight expression:"
            call config%pn_weight%write (u)
        end if
    else
        call write_separator (u)
        write (u, "(3x,A)") "Expressions (cut, scales, weight): [not shown]"
    end if
    if (config%md5sum /= "") then
        call write_separator (u)
        write (u, "(3x,A,A,A)") "MD5 sum (config) = '", config%md5sum, "'"
    end if
end subroutine process_config_data_write

```

Initialize. We use information from the process metadata and from the process library, given the process ID. We also store the currently active OS data set.

Using the model name that the library gives us, we try to load the model here and store the pointer in the configuration data.

```

<Processes: process config data: TBP>+≡
  procedure :: init => process_config_data_init

<Processes: procedures>+≡
  subroutine process_config_data_init (config, meta, os_data)
    class(process_config_data_t), intent(out) :: config
    type(process_metadata_t), intent(in) :: meta
    type(os_data_t), intent(in) :: os_data
    type(string_t) :: filename
    config%n_in = meta%lib%get_n_in (meta%id)
    config%n_components = size (meta%component_id)
    config%os_data = os_data
    config%model_name = meta%lib%get_model_name (meta%id)
    if (config%model_name /= "") then
      filename = config%model_name // ".mdl"
      call model_list_read_model (config%model_name, &
        filename, config%os_data, config%model)
    end if
  end subroutine process_config_data_init

```

Compute the MD5 sum of the configuration data. This encodes, in particular, the model and the expressions for cut, scales, weight, etc. It should not contain the IDs and number of components, etc., since the MD5 sum should be useful for integrating individual components.

This is done only once. If the MD5 sum is nonempty, the calculation is skipped.

```

<Processes: process config data: TBP>+≡
  procedure :: compute_md5sum => process_config_data_compute_md5sum

<Processes: procedures>+≡
  subroutine process_config_data_compute_md5sum (config)
    class(process_config_data_t), intent(inout) :: config
    integer :: u
    if (config%md5sum == "") then
      u = free_unit ()
      open (u, status = "scratch", action = "readwrite")
      call config%write (u, counters = .false., os_data = .false., &
        model = .true., expressions = .true.)
      rewind (u)
      config%md5sum = md5sum (u)
      close (u)
    end if
  end subroutine process_config_data_compute_md5sum

```

### 15.5.4 Beam configuration

The object `data` holds all details about the initial beam configuration. The allocatable array `sf` holds the structure-function configuration blocks. There are `n_strfun` entries in the structure-function chain (not counting the initial beam object). We maintain `n_channel` independent parameterizations of this chain. If this is greater than zero, we need a multi-channel sampling algorithm, where for each point one channel is selected to generate kinematics.

The number of parameters that are required for generating a structure-function chain is `n_sfpar`.

The flag `azimuthal_dependence` tells whether the process setup is symmetric about the beam axis in the c.m. system. This implies that there is no transversal beam polarization. The flag `lab_is_cm_frame` is obvious.

```

<Processes: process part types>+=
  type :: process_beam_config_t
    private
    type (beam_data_t) :: data
    integer :: n_strfun = 0
    integer :: n_channel = 1
    integer :: n_sfpar = 0
    type (sf_config_t), dimension(:), allocatable :: sf
    type (sf_channel_t), dimension(:), allocatable :: sf_channel
    logical :: azimuthal_dependence = .false.
    logical :: lab_is_cm_frame = .true.
    character(32) :: md5sum = ""
  contains
    <Processes: process beam config: TBP>
  end type process_beam_config_t

```

Here we write beam data only if they are actually used.

```

<Processes: process beam config: TBP>=
  procedure :: write => process_beam_config_write

<Processes: procedures>+=
  subroutine process_beam_config_write (object, u)
    class (process_beam_config_t), intent(in) :: object
    integer, intent(in) :: u
    integer :: i, c
    call object%data%write (u)
    if (object%data%initialized) then
      write (u, "(3x,A,L1)") "Azimuthal dependence    = ", &
        object%azimuthal_dependence
      write (u, "(3x,A,L1)") "Lab frame is c.m. frame = ", &
        object%lab_is_cm_frame
      if (object%md5sum /= "") then
        write (u, "(3x,A,A,A)") "MD5 sum (beams/strf) = '", &
          object%md5sum, "'"
      end if
      if (allocated (object%sf)) then
        do i = 1, size (object%sf)
          call object%sf(i)%write (u)
        end do
        if (any_sf_channel_has_mapping (object%sf_channel)) then

```

```

        write (u, "(1x,A,L1)") "Structure-function mappings per channel:"
        do c = 1, object%n_channel
            write (u, "(3x,I0,':')", advance="no") c
            call object%sf_channel(c)%write (u)
        end do
    end if
end if
end if
end if
end subroutine process_beam_config_write

```

The beam data have a finalizer. We assume that there is none for the structure-function data.

```

<Processes: process beam config: TBP>+≡
    procedure :: final => process_beam_config_final

<Processes: procedures>+≡
    subroutine process_beam_config_final (object)
        class(process_beam_config_t), intent(inout) :: object
        call beam_data_final (object%data)
    end subroutine process_beam_config_final

```

Initialize the beam setup with a given beam data object.

```

<Processes: process beam config: TBP>+≡
    procedure :: init_beam_data => process_beam_config_init_beam_data

<Processes: procedures>+≡
    subroutine process_beam_config_init_beam_data (beam_config, beam_data)
        class(process_beam_config_t), intent(out) :: beam_config
        type(beam_data_t), intent(in) :: beam_data
        beam_config%data = beam_data
    end subroutine process_beam_config_init_beam_data

```

Initialize the beam setup for a scattering process with trivial kinematics and polarization. We only need the c.m. energy and the incoming-state flavor combination.

```

<Processes: process beam config: TBP>+≡
    procedure :: init_scattering => process_beam_config_init_scattering

<Processes: procedures>+≡
    subroutine process_beam_config_init_scattering (beam_config, flv_in, sqrts)
        class(process_beam_config_t), intent(out) :: beam_config
        type(flavor_t), dimension(2), intent(in) :: flv_in
        real(default), intent(in) :: sqrts
        call beam_data_init_sqrts (beam_config%data, sqrts, flv_in)
    end subroutine process_beam_config_init_scattering

```

Print an informative message.

```

<Processes: process beam config: TBP>+≡
    procedure :: startup_message => process_beam_config_startup_message

```



```

<Processes: procedures>+≡
subroutine process_beam_config_startup_message (beam_config, unit)
  class(process_beam_config_t), intent(in) :: beam_config
  integer, intent(in), optional :: unit
  associate (data => beam_config%data)
    select case (data%n)
    case (1)
      write (msg_buffer, "(A,1x,A)") &
        "Beam:", char (flavor_get_name (data%flv(1)))
      call msg_message (unit = unit)
      write (msg_buffer, "(A,1x,A,ES10.3,1x,A)") &
        "Beam:", &
        "m =", data%sqrts, "GeV"
      call msg_message (unit = unit)
    case (2)
      write (msg_buffer, "(A,2(1x,A))") &
        "Beams:", &
        char (flavor_get_name (data%flv(1))), &
        char (flavor_get_name (data%flv(2)))
      call msg_message (unit = unit)
      write (msg_buffer, "(A,1x,A,ES10.3,1x,A)") &
        "Beams:", &
        "sqrts =", data%sqrts, "GeV"
      call msg_message (unit = unit)
    end select
  end associate
end subroutine process_beam_config_startup_message

```

Allocate the structure-function array.

Note: The direct source-allocation of beam\_config%sf is more elegant but causes memory corruption in gfortran 4.6.3.

```

<Processes: process beam config: TBP>+≡
  procedure :: init_sf_chain => process_beam_config_init_sf_chain

<Processes: procedures>+≡
subroutine process_beam_config_init_sf_chain (beam_config, sf_config)
  class(process_beam_config_t), intent(inout) :: beam_config
  type(sf_config_t), dimension(:), intent(in) :: sf_config
  integer :: i
  beam_config%n_strfun = size (sf_config)
  ! allocate (beam_config%sf (beam_config%n_strfun), source = sf_config)
  allocate (beam_config%sf (beam_config%n_strfun))
  do i = 1, beam_config%n_strfun
    call beam_config%sf(i)%init (sf_config(i)%i, sf_config(i)%data)
    beam_config%n_sfpar = beam_config%n_sfpar &
      + sf_config(i)%data%get_n_par ()
  end do
  call allocate_sf_channels (beam_config%sf_channel, &
    n_channel = beam_config%n_channel, &
    n_strfun = beam_config%n_strfun)
end subroutine process_beam_config_init_sf_chain

```

Set a structure-function mapping channel for an array of structure-function

entries, for a single channel. (The default is no mapping.)

```

(Processes: process beam config: TBP)+≡
  procedure :: set_sf_channel => process_beam_config_set_sf_channel

(Processes: procedures)+≡
  subroutine process_beam_config_set_sf_channel (beam_config, c, sf_channel)
    class(process_beam_config_t), intent(inout) :: beam_config
    integer, intent(in) :: c
    type(sf_channel_t), intent(in) :: sf_channel
    beam_config%sf_channel(c) = sf_channel
  end subroutine process_beam_config_set_sf_channel

```

Print an informative startup message.

```

(Processes: process beam config: TBP)+≡
  procedure :: sf_startup_message => process_beam_config_sf_startup_message

(Processes: procedures)+≡
  subroutine process_beam_config_sf_startup_message (beam_config, unit)
    class(process_beam_config_t), intent(in) :: beam_config
    integer, intent(in), optional :: unit
    if (beam_config%n_strfun > 0) then
      write (msg_buffer, "(A,3(1x,I0,1x,A))" &
        "Structure-function setup:", &
        beam_config%n_channel, "channels,", &
        beam_config%n_sfpar, "dimensions"
      call msg_message (unit = unit)
    end if
  end subroutine process_beam_config_sf_startup_message

```

Compute the MD5 sum for the complete beam setup. We rely on the default output of `write` to contain all relevant data.

This is done only once, when the MD5 sum is still empty.

```

(Processes: process beam config: TBP)+≡
  procedure :: compute_md5sum => process_beam_config_compute_md5sum

(Processes: procedures)+≡
  subroutine process_beam_config_compute_md5sum (beam_config)
    class(process_beam_config_t), intent(inout) :: beam_config
    integer :: u
    if (beam_config%md5sum == "") then
      u = free_unit ()
      open (u, status = "scratch", action = "readwrite")
      call beam_config%write (u)
      rewind (u)
      beam_config%md5sum = md5sum (u)
      close (u)
    end if
  end subroutine process_beam_config_compute_md5sum

```

### 15.5.5 Multi-channel integration

The `process_mci_entry_t` block contains, for each process component that is integrated independently, the configuration data for its MC input parameters.

Each input parameter set is handled by a `mci_t` integrator.

The MC input parameter set is broken down into the parameters required by the structure-function chain and the parameters required by the phase space of the elementary process.

The MD5 sum collects all information about the associated processes that may affect the integration. It does not contain the MCI object itself or integration results.

MC integration is organized in passes. Each pass may consist of several iterations, and for each iteration there is a number of calls. We store explicitly the values that apply to the current pass. Previous values are archived in the `results` object.

The `results` object records results, broken down in passes and iterations.

```

<Processes: process part types>+≡
  type :: process_mci_entry_t
    integer :: i_mci = 0
    integer, dimension(:), allocatable :: i_component
    integer :: process_type = PRC_UNKNOWN
    integer :: n_par = 0
    integer :: n_par_sf = 0
    integer :: n_par_phs = 0
    character(32) :: md5sum = ""
    integer :: pass = 0
    integer :: n_it = 0
    integer :: n_calls = 0
    class(mci_t), allocatable :: mci
    type(integration_results_t) :: results
  contains
    <Processes: process mci entry: TBP>
  end type process_mci_entry_t

```

Finalizer for the `mci` component.

```

<Processes: process mci entry: TBP>≡
  procedure :: final => process_mci_entry_final

<Processes: procedures>+≡
  subroutine process_mci_entry_final (object)
    class(process_mci_entry_t), intent(inout) :: object
    if (allocated (object%mci)) call object%mci%final ()
  end subroutine process_mci_entry_final

```

Output. Write pass/iteration information only if set (the pass index is nonzero). Write the MCI block only if it exists (for some self-tests it does not). Write results only if there are any.

```

<Processes: process mci entry: TBP>+≡
  procedure :: write => process_mci_entry_write

<Processes: procedures>+≡
  subroutine process_mci_entry_write (object, unit)
    class(process_mci_entry_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)

```

```

write (u, "(3x,A,I0)") "Associated components = ", object%i_component
write (u, "(3x,A,I0)") "MC input parameters = ", object%n_par
write (u, "(3x,A,I0)") "MC parameters (SF) = ", object%n_par_sf
write (u, "(3x,A,I0)") "MC parameters (PHS) = ", object%n_par_phs
if (object%pass > 0) then
  write (u, "(3x,A,I0)") "Current pass = ", object%pass
  write (u, "(3x,A,I0)") "Number of iterations = ", object%n_it
  write (u, "(3x,A,I0)") "Number of calls = ", object%n_calls
end if
if (object%md5sum /= "") then
  write (u, "(3x,A,A,A)") "MD5 sum (components) = ', object%md5sum, '"
end if
if (allocated (object%mc_i)) then
  call object%mc_i%write (u)
end if
if (object%results%exist ()) then
  call object%results%write (u)
end if
end subroutine process_mci_entry_write

```

Initialize. From the existing configuration, we obtain the number of channels and the number of parameters, separately for the structure-function chain and for the associated process component. We assume that the phase-space object has already been configured.

The allocation of the MCI integrator with the appropriate concrete type is the duty of the process core.

We assume that there is only one component associated with a MCI entry. This restriction should be relaxed.

```

<Processes: process mci entry: TBP>+≡
  procedure :: init => process_mci_entry_init

<Processes: procedures>+≡
  subroutine process_mci_entry_init &
    (mci_entry, process_type, i_mci, i_component, component, beam_config)
    class(process_mci_entry_t), intent(out) :: mci_entry
    integer, intent(in) :: process_type
    integer, intent(in) :: i_mci
    integer, intent(in) :: i_component
    type(process_component_t), intent(in), target :: component
    type(process_beam_config_t), intent(in) :: beam_config
    associate (phs_config => component%phs_config)
      mci_entry%i_mci = i_mci
      allocate (mci_entry%i_component (1))
      mci_entry%i_component(1) = i_component
      mci_entry%n_par_sf = beam_config%n_sfpar
      mci_entry%n_par_phs = phs_config%get_n_par ()
      mci_entry%n_par = mci_entry%n_par_sf + mci_entry%n_par_phs
      mci_entry%process_type = process_type
      if (allocated (component%mc_i_template)) then
        allocate (mci_entry%mc_i, source=component%mc_i_template)
        call mci_entry%mc_i%set_dimensions &
          (mci_entry%n_par, phs_config%get_n_channel ())
        call mci_entry%mc_i%declare_flat_dimensions &

```

```

        (phs_config%get_flat_dimensions ())
    if (phs_config%provides_equivalences) then
        call mci_entry%mci%declare_equivalences &
            (phs_config%channel, mci_entry%n_par_sf)
    end if
    if (phs_config%provides_chains) then
        call mci_entry%mci%declare_chains (phs_config%chain)
    end if
end if
end associate
call mci_entry%results%init (process_type)
end subroutine process_mci_entry_init

```

Compute a MD5 sum that summarizes all information for the associated process components. We take the process-configuration MD5 sum which represents parameters, cuts, etc., the MD5 sums for the process component definitions and their phase space objects (which should be configured), and the beam configuration MD5 sum.

Done only once, when the MD5 sum is still empty.

```

<Processes: process mci entry: TBP>+≡
    procedure :: compute_md5sum => process_mci_entry_compute_md5sum
<Processes: procedures>+≡
    subroutine process_mci_entry_compute_md5sum (mci_entry, &
        config, component, beam_config)
        class(process_mci_entry_t), intent(inout) :: mci_entry
        type(process_config_data_t), intent(in) :: config
        type(process_component_t), dimension(:), intent(in) :: component
        type(process_beam_config_t), intent(in) :: beam_config
        type(string_t) :: buffer
        integer :: i
        if (mci_entry%md5sum == "") then
            buffer = config%md5sum // beam_config%md5sum
            do i = 1, size (component)
                buffer = buffer // component(i)%config%get_md5sum () &
                    // component(i)%md5sum_phs
            end do
            mci_entry%md5sum = md5sum (char (buffer))
        end if
        if (allocated (mci_entry%mci)) then
            call mci_entry%mci%set_md5sum (mci_entry%md5sum)
        end if
    end subroutine process_mci_entry_compute_md5sum

```

Integrate. The instance should be initialized.

The `integrate` method counts as an integration pass; the pass count is increased by one. We transfer the pass parameters (number of iterations and number of calls) to the actual integration routine.

The `mci_entry` is `intent(inout)` because the integrator contains the random-number state.

Note: The results are written to screen and to logfile. This behavior is hardcoded.

```

<Processes: process mci entry: TBP>+≡

```

```

procedure :: integrate => process_mci_entry_integrate
procedure :: final_integration => process_mci_entry_final_integration

<Processes: procedures>+≡
  subroutine process_mci_entry_integrate (mci_entry, instance, n_it, n_calls, &
    adapt_grids, adapt_weights)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    type(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
    integer :: u_log
    u_log = logfile_unit ()
    call instance%choose_mci (mci_entry%i_mci)
    mci_entry%pass = mci_entry%pass + 1
    mci_entry%n_it = n_it
    mci_entry%n_calls = n_calls
    if (mci_entry%pass == 1) call mci_entry%mci%startup_message ()
    call mci_entry%results%display_init &
      (mci_entry%process_type, screen = .true., unit = u_log)
    call mci_entry%results%new_pass ()
    associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
      call mci_entry%mci%add_pass (adapt_grids, adapt_weights)
      call mci_entry%mci%integrate (mci_instance, instance, n_it, n_calls, &
        mci_entry%results)
    end associate
    call mci_entry%results%display_pass ()
  end subroutine process_mci_entry_integrate

  subroutine process_mci_entry_final_integration (mci_entry)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    call mci_entry%results%display_final ()
  end subroutine process_mci_entry_final_integration

```

Initialize and finalize event generation. This calls the the associated initializers and finalizers for the MCI entry, respectively.

Note: For the test integrator, this does nothing. It is relevant for the VAMP integrator.

```

<Processes: process mci entry: TBP>+≡
  procedure :: init_simulation => process_mci_entry_init_simulation
  procedure :: final_simulation => process_mci_entry_final_simulation

<Processes: procedures>+≡
  subroutine process_mci_entry_init_simulation (mci_entry, instance)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    type(process_instance_t), intent(inout) :: instance
    associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
      call mci_entry%mci%init_simulation (mci_instance)
    end associate
  end subroutine process_mci_entry_init_simulation

  subroutine process_mci_entry_final_simulation (mci_entry, instance)
    class(process_mci_entry_t), intent(in) :: mci_entry

```

```

type(process_instance_t), intent(inout) :: instance
associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
    call mci_entry%mci%final_simulation (mci_instance)
end associate
end subroutine process_mci_entry_final_simulation

```

Generate an event. The instance should be initialized, otherwise event generation is directed by the mci integrator subobject. The integrator instance is contained in a mci\_work subobject of the process instance, which simultaneously serves as the sampler object. (We avoid the anti-aliasing rules if we assume that the sampling itself does not involve the integrator instance contained in the process instance.)

```

<Processes: process mci entry: TBP>+≡
    procedure :: generate_weighted_event => &
        process_mci_entry_generate_weighted_event
    procedure :: generate_unweighted_event => &
        process_mci_entry_generate_unweighted_event
    procedure :: recover_event => process_mci_entry_recover_event

<Processes: procedures>+≡
    subroutine process_mci_entry_generate_weighted_event (mci_entry, instance)
        class(process_mci_entry_t), intent(inout) :: mci_entry
        type(process_instance_t), intent(inout) :: instance
        call instance%choose_mci (mci_entry%i_mci)
        associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
            call mci_entry%mci%generate_weighted_event (mci_instance, instance)
        end associate
    end subroutine process_mci_entry_generate_weighted_event

    subroutine process_mci_entry_generate_unweighted_event (mci_entry, instance)
        class(process_mci_entry_t), intent(inout) :: mci_entry
        type(process_instance_t), intent(inout) :: instance
        call instance%choose_mci (mci_entry%i_mci)
        associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
            call mci_entry%mci%generate_unweighted_event (mci_instance, instance)
        end associate
    end subroutine process_mci_entry_generate_unweighted_event

    subroutine process_mci_entry_recover_event (mci_entry, instance, i_term)
        class(process_mci_entry_t), intent(inout) :: mci_entry
        type(process_instance_t), intent(inout) :: instance
        integer, intent(in) :: i_term
        integer :: channel
        mci_entry%i_mci = instance%i_mci
        channel = instance%get_channel ()
        associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
            call mci_instance%fetch (instance, channel)
        end associate
    end subroutine process_mci_entry_recover_event

```

Extract results.

```

<Processes: process mci entry: TBP>+≡
    procedure :: get_integral => process_mci_entry_get_integral

```

```

    procedure :: get_error => process_mci_entry_get_error
  (Processes: procedures)+≡
    function process_mci_entry_get_integral (mci_entry) result (integral)
      class(process_mci_entry_t), intent(in) :: mci_entry
      real(default) :: integral
      integral = mci_entry%results%get_integral ()
    end function process_mci_entry_get_integral

    function process_mci_entry_get_error (mci_entry) result (error)
      class(process_mci_entry_t), intent(in) :: mci_entry
      real(default) :: error
      error = mci_entry%results%get_error ()
    end function process_mci_entry_get_error

```

Return the MCI checksum. This may be the one used for configuration, but may also incorporate results, if they change the state of the integrator (adaptation).

```

  (Processes: process mci entry: TBP)+≡
    procedure :: get_md5sum => process_mci_entry_get_md5sum
  (Processes: procedures)+≡
    function process_mci_entry_get_md5sum (entry) result (md5sum)
      class(process_mci_entry_t), intent(in) :: entry
      character(32) :: md5sum
      md5sum = entry%mci%get_md5sum ()
    end function process_mci_entry_get_md5sum

```

### 15.5.6 Process Components

A process component is an individual contribution to a process (scattering or decay) which needs not be physical. The sum over all components should be physical.

The `index` identifies this component within its parent process.

The actual process component is stored in the `core` subobject. We use a polymorphic subobject instead of an extension of `process_component_t`, because the individual entries in the array of process components have different types. In short, `process_component_t` is a wrapper for the actual process variants.

The index array `i_term` points to the individual terms generated by this component. The indices refer to the parent process.

The index `i_mci` is the index of the MC integrator and parameter set which are associated to this process component.

```

  (Processes: process part types)+≡
    type :: process_component_t
      private
      type(process_component_def_t), pointer :: config => null ()
      integer :: index = 0
      class(prc_core_t), allocatable :: core
      class(mci_t), allocatable :: mci_template
      integer, dimension(:), allocatable :: i_term
      integer :: i_mci = 0

```



```

        class(phs_config_t), allocatable :: phs_config
        character(32) :: md5sum_phs = ""
    contains
        <Processes: process component: TBP>
    end type process_component_t

```

Finalizer. The MCI template may (potentially) need a finalizer. The process configuration finalizer may include closing an open scratch file.

```

<Processes: process component: TBP>≡
    procedure :: final => process_component_final

<Processes: procedures>+≡
    subroutine process_component_final (object)
        class(process_component_t), intent(inout) :: object
        if (allocated (object%mci_template)) then
            call object%mci_template%final ()
        end if
        if (allocated (object%phs_config)) then
            call object%phs_config%final ()
        end if
    end subroutine process_component_final

```

The meaning of *verbose* depends on the process variant.

```

<Processes: process component: TBP>+≡
    procedure :: write => process_component_write

<Processes: procedures>+≡
    subroutine process_component_write (object, unit)
        class(process_component_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        if (allocated (object%core)) then
            write (u, "(1x,A,I0)") "Component #", object%index
            if (associated (object%config)) then
                call object%config%write (u)
                if (object%md5sum_phs /= "") then
                    write (u, "(3x,A,A,A)") "MD5 sum (phs)      = '", &
                        object%md5sum_phs, "'"
                end if
            end if
            write (u, "(1x,A)") "Process core:"
            call object%core%write (u)
        else
            write (u, "(1x,A)") "Process component: [not allocated]"
        end if
        write (u, "(1x,A)") "Referenced data:"
        if (allocated (object%i_term)) then
            write (u, "(3x,A,999(1x,I0))") "Terms                =", &
                object%i_term
        else
            write (u, "(3x,A)") "Terms                = [undefined]"
        end if
        if (object%i_mci /= 0) then

```

```

        write (u, "(3x,A,I0)") "MC dataset"           = ", object%i_mci
    else
        write (u, "(3x,A)") "MC dataset"             = [undefined]"
    end if
    if (allocated (object%phs_config)) then
        call object%phs_config%write (u)
    end if
end subroutine process_component_write

```

Initialize the component.

*(Processes: process component: TBP)+≡*

```

    procedure :: init => process_component_init

```

*(Processes: procedures)+≡*

```

    subroutine process_component_init (component, &
        i_component, meta, core_template, mci_template, phs_config_template)
    class(process_component_t), intent(out) :: component
    integer, intent(in) :: i_component
    type(process_metadata_t), intent(in) :: meta
    class(prc_core_t), intent(in), allocatable :: core_template
    class(mci_t), intent(in), allocatable :: mci_template
    class(phs_config_t), intent(in), allocatable :: phs_config_template
    component%index = i_component
    component%config => meta%lib%get_component_def_ptr (meta%id, i_component)
    allocate (component%core, source=core_template)
    call component%core%init (component%config%get_core_def_ptr (), &
        meta%lib, meta%id, i_component)
    if (allocated (mci_template)) &
        allocate (component%mci_template, source=mci_template)
    allocate (component%phs_config, source=phs_config_template)
    call component%phs_config%init (component%core%data)
end subroutine process_component_init

```

Finalize the phase-space configuration.

*(Processes: process component: TBP)+≡*

```

    procedure :: configure_phs => process_component_configure_phs

```

*(Processes: procedures)+≡*

```

    subroutine process_component_configure_phs &
        (component, sqrts, beam_config, rebuild)
    class(process_component_t), intent(inout) :: component
    real(default), intent(in) :: sqrts
    type(process_beam_config_t), intent(in) :: beam_config
    logical, intent(in), optional :: rebuild
    logical :: no_strfun
    no_strfun = beam_config%n_strfun == 0
    call component%phs_config%configure (sqrts, &
        azimuthal_dependence = beam_config%azimuthal_dependence, &
        sqrts_fixed = no_strfun, &
        cm_frame = beam_config%lab_is_cm_frame .and. no_strfun, &
        rebuild = rebuild)
    call component%phs_config%startup_message ()
end subroutine process_component_configure_phs

```

The process component possesses two MD5 sums: the checksum of the component definition, which should be available when the component is initialized, and the phase-space MD5 sum, which is available after configuration.

```

(Processes: process component: TBP)+≡
  procedure :: compute_md5sum => process_component_compute_md5sum

(Processes: procedures)+≡
  subroutine process_component_compute_md5sum (component)
    class(process_component_t), intent(inout) :: component
    component%md5sum_phs = component%phs_config%get_md5sum ()
  end subroutine process_component_compute_md5sum

```

Match phase-space channels with structure-function channels, where applicable. This calls a method of the `phs_config` phase-space implementation.

```

(Processes: process component: TBP)+≡
  procedure :: match_channels => process_component_match_channels

(Processes: procedures)+≡
  subroutine process_component_match_channels (component, beam_config)
    class(process_component_t), intent(inout) :: component
    type(process_beam_config_t), intent(in) :: beam_config
    call component%phs_config%match_channels &
      (beam_config%sf, beam_config%sf_channel)
  end subroutine process_component_match_channels

```

Return the incoming flavor combination as an integer array. By calling this we assume that the flavor combination is unique. If it is not, return an undefined flavor state. We also assume that the size of the

```

(Processes: process component: TBP)+≡
  procedure :: get_flv_state_in => process_component_get_flv_state_in

(Processes: procedures)+≡
  function process_component_get_flv_state_in (component) result (flv_state_in)
    class(process_component_t), intent(in) :: component
    integer, dimension(:), allocatable :: flv_state_in
    integer :: i
    associate (data => component%core%data)
      allocate (flv_state_in (data%n_in))
      do i = 1, data%n_in
        if (all (data%flv_state(i,:) == data%flv_state(i,1))) then
          flv_state_in(i) = data%flv_state(i,1)
        else
          flv_state_in(i) = UNDEFINED
        end if
      end do
    end associate
  end function process_component_get_flv_state_in

```

## 15.5.7 Process terms

For straightforward tree-level calculations, each process component corresponds to a unique elementary interaction. However, in the case of NLO calcula-

tions with subtraction terms, a process component may split into several separate contributions to the scattering, which are qualified by interactions with distinct kinematics and particle content. We represent their configuration as `process_term_t` objects, the actual instances will be introduced below as `term_instance_t`. In any case, the process term contains an elementary interaction with a definite quantum-number and momentum content.

The index `i_term_global` identifies the term relative to the process.

The index `i_component` identifies the process component which generates this term, relative to the parent process.

The index `i_term` identifies the term relative to the process component (not the process).

The `data` subobject holds all process constants.

The number of allowed flavor/helicity/color combinations is stored as `n_allowed`. This is the total number of independent entries in the density matrix. For each combination, the index of the flavor, helicity, and color state is stored in the arrays `flv`, `hel`, and `col`, respectively.

The flag `rearrange` is true if we need to rearrange the particles of the hard interaction, to obtain the effective parton state.

The interaction `int` holds the quantum state for the (resolved) hard interaction, the parent-child relations of the particles, and their momenta. The momenta are not filled yet; this is postponed to copies of `int` which go into the process instances.

If recombination is in effect, we should allocate `int_eff` to describe the rearranged partonic state.

```

<Processes: process part types>+=
  type :: process_term_t
    integer :: i_term_global = 0
    integer :: i_component = 0
    integer :: i_term = 0
    integer :: n_allowed = 0
    type(process_constants_t) :: data
    real(default) :: alpha_s = 0
    integer, dimension(:), allocatable :: flv, hel, col
    logical :: rearrange = .false.
    type(interaction_t) :: int
    type(interaction_t), pointer :: int_eff => null ()
  contains
    <Processes: process term: TBP>
  end type process_term_t

```

For the output, we skip the process constants and the tables of allowed quantum numbers. Those can also be read off from the interaction object.

```

<Processes: process term: TBP>=
  procedure :: write => process_term_write

<Processes: procedures>+=
  subroutine process_term_write (term, unit)
    class(process_term_t), intent(in) :: term
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(1x,A,I0)") "Term #", term%i_term_global

```

```

write (u, "(3x,A,I0)") "Process component index      = ", &
    term%i_component
write (u, "(3x,A,I0)") "Term index w.r.t. component = ", &
    term%i_term
write (u, "(3x,A,L1)") "Rearrange partons          = ", &
    term%rearrange
call write_separator (u)
write (u, "(1x,A)") "Hard interaction:"
call write_separator (u)
call interaction_write (term%int, u)
if (term%rearrange) then
    call write_separator (u)
    write (u, "(1x,A)") "Rearranged hard interaction:"
    call write_separator (u)
    call interaction_write (term%int_eff, u)
end if
end subroutine process_term_write

```

Finalizer: the `int` and potentially `int_eff` components have a finalizer that we must call.

```

<Processes: process term: TBP>+≡
    procedure :: final => process_term_final

<Processes: procedures>+≡
    subroutine process_term_final (term)
        class(process_term_t), intent(inout) :: term
        call interaction_final (term%int)
        if (term%rearrange) then
            call interaction_final (term%int_eff)
            deallocate (term%int_eff)
        end if
    end subroutine process_term_final

```

Initialize the term. We copy the process constants from the `core` object and set up the `int` hard interaction accordingly.

The `alpha_s` value is useful for writing external event records. This is the constant value which may be overridden by a event-specific running value. If the model does not contain the strong coupling, the value is zero.

Note: `intent(inout)` for `term` is a workaround for a NAG compiler bug.

```

<Processes: process term: TBP>+≡
    procedure :: init => process_term_init

<Processes: procedures>+≡
    subroutine process_term_init &
        (term, i_term_global, i_component, i_term, core, model)
        class(process_term_t), intent(inout), target :: term
        integer, intent(in) :: i_term_global
        integer, intent(in) :: i_component
        integer, intent(in) :: i_term
        class(prc_core_t), intent(in) :: core
        type(model_t), intent(in), target :: model
        type(var_list_t), pointer :: var_list
        term%i_term_global = i_term_global
    end subroutine process_term_init

```

```

term%i_component = i_component
term%i_term = i_term
call core%get_constants (term%data, i_term)
var_list => model_get_var_list_ptr (model)
if (var_list_exists (var_list, var_str ("alphas"))) then
    term%alpha_s = var_list_get_rval (var_list, var_str ("alphas"))
else
    term%alpha_s = -1
end if
call term%setup_interaction (core, model)
!   if (term%rearrange) then
!       call term%setup_effective_interaction (core, term%int, term%int_eff)
!   end if
end subroutine process_term_init

```

We fetch the process constants which determine the quantum numbers and use those to create the interaction. The interaction contains incoming and outgoing particles, no virtuals. The incoming particles are parents of the outgoing ones.

Keeping previous WHIZARD conventions, we invert the color assignment (but not flavor or helicity) for the incoming particles. When the color-flow square matrix is evaluated, this inversion is done again, so in the color-flow sequence we get the color assignments of the matrix element.

*(Processes: process term: TBP)+≡*

```

procedure :: setup_interaction => process_term_setup_interaction

```

*(Processes: procedures)+≡*

```

subroutine process_term_setup_interaction (term, core, model)
    class(process_term_t), intent(inout) :: term
    class(prc_core_t), intent(in) :: core
    type(model_t), intent(in), target :: model
    integer :: n_tot
    type(flavor_t), dimension(:), allocatable :: flv
    type(color_t), dimension(:), allocatable :: col
    type(helicity_t), dimension(:), allocatable :: hel
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: i, n, f, h, c
    associate (data => term%data)
        n_tot = data%n_in + data%n_out
        n = 0
        do f = 1, data%n_flv
            do h = 1, data%n_hel
                do c = 1, data%n_col
                    if (core%is_allowed (term%i_term, f, h, c)) n = n + 1
                end do
            end do
        end do
        allocate (term%flv (n), term%col (n), term%hel (n))
        term%n_allowed = n
        allocate (flv (n_tot), col (n_tot), hel (n_tot))
        allocate (qn (n_tot))
        call interaction_init &
            (term%int, data%n_in, 0, data%n_out, set_relations=.true.)
        i = 0
    end associate
end subroutine process_term_setup_interaction

```

```

do f = 1, data%n_flv
  do h = 1, data%n_hel
    do c = 1, data%n_col
      if (core%is_allowed (term%i_term, f, h, c)) then
        i = i + 1
        term%flv(i) = f
        term%hel(i) = h
        term%col(i) = c
        call flavor_init (flv, data%flv_state(:,f), model)
        call color_init_from_array (col, data%col_state(:, :, c), &
          data%ghost_flag(:, c))
        call color_invert (col(:data%n_in))
        call helicity_init (hel, data%hel_state(:, h))
        call quantum_numbers_init (qn, flv, col, hel)
        call interaction_add_state (term%int, qn)
      end if
    end do
  end do
end do
call interaction_freeze (term%int)
end associate
end subroutine process_term_setup_interaction

```

### 15.5.8 Default Iterations

If the user does not specify the passes and iterations for integration, we should be able to give reasonable defaults. These depend on the process, therefore we implement the following procedures as methods of the process object. The algorithm is not very sophisticated yet, it may be improved by looking at the process in more detail.

We investigate only the first process component, assuming that it characterizes the complexity of the process reasonable well.

The number of passes is limited to two: one for adaption, one for integration.

*(Processes: process: TBP)+≡*

```

procedure :: get_n_pass_default => process_get_n_pass_default
procedure :: adapt_grids_default => process_adapt_grids_default
procedure :: adapt_weights_default => process_adapt_weights_default

```

*(Processes: procedures)+≡*

```

function process_get_n_pass_default (process) result (n_pass)
  class(process_t), intent(in) :: process
  integer :: n_pass
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (n_eff)
  case (1)
    n_pass = 1
  case default
    n_pass = 2
  end select
end function

```

```

end function process_get_n_pass_default

function process_adapt_grids_default (process, pass) result (flag)
  class(process_t), intent(in) :: process
  integer, intent(in) :: pass
  logical :: flag
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (n_eff)
  case (1)
    flag = .false.
  case default
    select case (pass)
    case (1); flag = .true.
    case (2); flag = .false.
    end select
  end select
end function process_adapt_grids_default

function process_adapt_weights_default (process, pass) result (flag)
  class(process_t), intent(in) :: process
  integer, intent(in) :: pass
  logical :: flag
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (n_eff)
  case (1)
    flag = .false.
  case default
    select case (pass)
    case (1); flag = .true.
    case (2); flag = .false.
    end select
  end select
end function process_adapt_weights_default

```

The number of iterations and calls per iteration depends on the number of outgoing particles.

```

<Processes: process: TBP>+≡
  procedure :: get_n_it_default => process_get_n_it_default
  procedure :: get_n_calls_default => process_get_n_calls_default

<Processes: procedures>+≡
  function process_get_n_it_default (process, pass) result (n_it)
    class(process_t), intent(in) :: process
    integer, intent(in) :: pass
    integer :: n_it
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config

```



```

n_eff = config%get_n_tot () - 2
select case (pass)
case (1)
  select case (n_eff)
  case (1);   n_it = 1
  case (2);   n_it = 3
  case (3);   n_it = 5
  case (4:5); n_it = 10
  case (6);   n_it = 15
  case (7:);  n_it = 20
  end select
case (2)
  select case (n_eff)
  case (:3);  n_it = 3
  case (4:);  n_it = 5
  end select
end select
end function process_get_n_it_default

function process_get_n_calls_default (process, pass) result (n_calls)
  class(process_t), intent(in) :: process
  integer, intent(in) :: pass
  integer :: n_calls
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (pass)
  case (1)
    select case (n_eff)
    case (1);   n_calls = 100
    case (2);   n_calls = 1000
    case (3);   n_calls = 5000
    case (4);   n_calls = 10000
    case (5);   n_calls = 20000
    case (6:);  n_calls = 50000
    end select
  case (2)
    select case (n_eff)
    case (:3);  n_calls = 10000
    case (4);   n_calls = 20000
    case (5);   n_calls = 50000
    case (6);   n_calls = 100000
    case (7:);  n_calls = 200000
    end select
  end select
end function process_get_n_calls_default

```

### 15.5.9 Constant process data

The following methods return basic process data that stay constant after initialization.

The process and IDs.

```

<Processes: process: TBP>+=
  procedure :: get_id => process_get_id
  procedure :: get_run_id => process_get_run_id
  procedure :: get_library_name => process_get_library_name

<Processes: procedures>+=
  function process_get_id (process) result (id)
    class(process_t), intent(in) :: process
    type(string_t) :: id
    id = process%meta%id
  end function process_get_id

  function process_get_run_id (process) result (id)
    class(process_t), intent(in) :: process
    type(string_t) :: id
    id = process%meta%run_id
  end function process_get_run_id

  function process_get_library_name (process) result (id)
    class(process_t), intent(in) :: process
    type(string_t) :: id
    id = process%meta%lib%get_name ()
  end function process_get_library_name

```

The number of incoming particles.

```

<Processes: process: TBP>+=
  procedure :: get_n_in => process_get_n_in

<Processes: procedures>+=
  function process_get_n_in (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%config%n_in
  end function process_get_n_in

```

The number of MCI data sets.

```

<Processes: process: TBP>+=
  procedure :: get_n_mci => process_get_n_mci

<Processes: procedures>+=
  function process_get_n_mci (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%config%n_mci
  end function process_get_n_mci

```

The number of process components, total.

```

<Processes: process: TBP>+=
  procedure :: get_n_components => process_get_n_components

```

```

<Processes: procedures>+≡
function process_get_n_components (process) result (n)
  class(process_t), intent(in) :: process
  integer :: n
  n = process%meta%n_components
end function process_get_n_components

```

Return the indices of the components that belong to a specific MCI entry.

```

<Processes: process: TBP>+≡
  procedure :: get_i_component => process_get_i_component

<Processes: procedures>+≡
  subroutine process_get_i_component (process, i_mci, i_component)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_mci
    integer, dimension(:), intent(out), allocatable :: i_component
    associate (mci_entry => process%mci_entry(i_mci))
      allocate (i_component (size (mci_entry%i_component)))
      i_component = mci_entry%i_component
    end associate
  end subroutine process_get_i_component

```

Return the ID of a specific component.

```

<Processes: process: TBP>+≡
  procedure :: get_component_id => process_get_component_id

<Processes: procedures>+≡
  function process_get_component_id (process, i_component) result (id)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_component
    type(string_t) :: id
    id = process%meta%component_id(i_component)
  end function process_get_component_id

```

Return a pointer to the definition of a specific component.

```

<Processes: process: TBP>+≡
  procedure :: get_component_def_ptr => process_get_component_def_ptr

<Processes: procedures>+≡
  function process_get_component_def_ptr (process, i_component) result (ptr)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_component
    type(process_component_def_t), pointer :: ptr
    ptr => process%meta%lib%get_component_def_ptr (process%meta%id, i_component)
  end function process_get_component_def_ptr

```

These procedures extract and restore (by transferring the allocation) the process core. This is useful for changing process parameters from outside this module.

```

<Processes: process: TBP>+≡
  procedure :: extract_component_core => process_extract_component_core
  procedure :: restore_component_core => process_restore_component_core

```

```

<Processes: procedures>+≡
  subroutine process_extract_component_core (process, i_component, core)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_component
    class(prc_core_t), intent(inout), allocatable :: core
    call move_alloc (from = process%component(i_component)%core, to = core)
  end subroutine process_extract_component_core

  subroutine process_restore_component_core (process, i_component, core)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_component
    class(prc_core_t), intent(inout), allocatable :: core
    call move_alloc (from = core, to = process%component(i_component)%core)
  end subroutine process_restore_component_core

```

The block of process constants.

```

<Processes: process: TBP>+≡
  procedure :: get_constants => process_get_constants

<Processes: procedures>+≡
  function process_get_constants (process, i) result (data)
    class(process_t), intent(in) :: process
    integer, intent(in) :: i
    type(process_constants_t) :: data
    data = process%component(i)%core%data
  end function process_get_constants

```

The nominal process energy.

```

<Processes: process: TBP>+≡
  procedure :: get_sqrts => process_get_sqrts

<Processes: procedures>+≡
  function process_get_sqrts (process) result (sqrts)
    class(process_t), intent(in) :: process
    real(default) :: sqrts
    sqrts = beam_data_get_sqrts (process%beam_config%data)
  end function process_get_sqrts

```

Pointer to the beam data object.

```

<Processes: process: TBP>+≡
  procedure :: get_beam_data_ptr => process_get_beam_data_ptr

<Processes: procedures>+≡
  function process_get_beam_data_ptr (process) result (beam_data)
    class(process_t), intent(in), target :: process
    type(beam_data_t), pointer :: beam_data
    beam_data => process%beam_config%data
  end function process_get_beam_data_ptr

```

Pointer to the process variable list.

```

<Processes: process: TBP>+≡
  procedure :: get_var_list_ptr => process_get_var_list_ptr

```

```

<Processes: procedures>+≡
function process_get_var_list_ptr (process) result (ptr)
  class(process_t), intent(in), target :: process
  type(var_list_t), pointer :: ptr
  ptr => process%meta%var_list
end function process_get_var_list_ptr

```

Pointer to the common model.

```

<Processes: process: TBP>+≡
procedure :: get_model_ptr => process_get_model_ptr

```

```

<Processes: procedures>+≡
function process_get_model_ptr (process) result (ptr)
  class(process_t), intent(in) :: process
  type(model_t), pointer :: ptr
  ptr => process%config%model
end function process_get_model_ptr

```

### 15.5.10 Compute an amplitude

Each process variant should allow for computing an amplitude value directly, without generating a process instance.

The process component is selected by the index *i*. The term within the process component is selected by *j*. The momentum combination is transferred as the array *p*. The function sets the specific quantum state via the indices of a flavor *f*, helicity *h*, and color *c* combination. Each index refers to the list of flavor, helicity, and color states, respectively, as stored in the process data.

Optionally, we may set factorization and renormalization scale. If unset, the partonic c.m. energy is inserted.

The function checks arguments for validity. For invalid arguments (quantum states), we return zero.

```

<Processes: process: TBP>+≡
procedure :: compute_amplitude => process_compute_amplitude

<Processes: procedures>+≡
function process_compute_amplitude &
  (process, i, j, p, f, h, c, fac_scale, ren_scale) result (amp)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i, j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in), optional :: fac_scale, ren_scale
  real(default) :: fscale, rscale
  complex(default) :: amp
  amp = 0
  if (0 < i .and. i <= process%meta%n_components) then
    associate (data => process%component(i)%core%data)
      if (size(p) == data%n_in + data%n_out &
        .and. 0 < f .and. f <= data%n_flv &
        .and. 0 < h .and. h <= data%n_hel &
        .and. 0 < c .and. c <= data%n_col) then
        if (present(fac_scale)) then

```

```

        fscale = fac_scale
    else
        fscale = sum (p(data%n_in+1:)) ** 1
    end if
    if (present (ren_scale)) then
        rscale = ren_scale
    else
        rscale = fscale
    end if
    amp = process%component(i)%core% &
        compute_amplitude (j, p, f, h, c, fscale, rscale)
    end if
end associate
end if
end function process_compute_amplitude

```

### 15.5.11 Process instances

#### Kinematics instance

In this data type we combine all objects (instances) necessary for generating (or recovering) a kinematical configuration. The components work together as an implementation of multi-channel phase space.

**sf\_chain** is an instance of the structure-function chain. It is used both for generating kinematics and, after the proper scale has been determined, evaluating the structure function entries.

**phs** is an instance of the phase space for the elementary process.

The array **f** contains the products of the Jacobians that originate from parameter mappings in the structure-function chain or in the phase space. We allocate this explicitly if either **sf\_chain** or **phs** are explicitly allocated, otherwise we can take over a pointer.

All components are implemented as pointers to (anonymous) targets. For each component, there is a flag that tells whether this component is to be regarded as a proper component ('owned' by the object) or as a pointer.

*(Processes: types)*≡

```

type :: kinematics_t
    integer :: n_in = 0
    integer :: n_channel = 0
    integer :: selected_channel = 0
    type(sf_chain_instance_t), pointer :: sf_chain => null ()
    class(phs_t), pointer :: phs => null ()
    real(default), dimension(:), pointer :: f => null ()
    real(default) :: phs_factor
    logical :: sf_chain_allocated = .false.
    logical :: phs_allocated = .false.
    logical :: f_allocated = .false.
contains
    (Processes: kinematics: TBP)
end type kinematics_t

```

Output. Show only those components which are marked as owned.

```

<Processes: kinematics: TBP>≡
  procedure :: write => kinematics_write

<Processes: procedures>+≡
  subroutine kinematics_write (object, unit)
    class(kinematics_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, c
    u = output_unit (unit)
    if (object%f_allocated) then
      write (u, "(1x,A)") "Flux * PHS volume:"
      write (u, "(2x,ES19.12)") object%phs_factor
      write (u, "(1x,A)") "Jacobian factors per channel:"
      do c = 1, size (object%f)
        write (u, "(3x,I0,':',1x,ES13.7)", advance="no") c, object%f(c)
        if (c == object%selected_channel) then
          write (u, "(1x,A)") "[selected]"
        else
          write (u, *)
        end if
      end do
    end if
    if (object%sf_chain_allocated) then
      call write_separator (u)
      call object%sf_chain%write (u)
    end if
    if (object%phs_allocated) then
      call write_separator (u)
      call object%phs%write (u)
    end if
  end subroutine kinematics_write

```

Finalizer. Delete only those components which are marked as owned.

```

<Processes: kinematics: TBP>+≡
  procedure :: final => kinematics_final

<Processes: procedures>+≡
  subroutine kinematics_final (object)
    class(kinematics_t), intent(inout) :: object
    if (object%sf_chain_allocated) then
      call object%sf_chain%final ()
      deallocate (object%sf_chain)
      object%sf_chain_allocated = .false.
    end if
    if (object%phs_allocated) then
      call object%phs%final ()
      deallocate (object%phs)
      object%phs_allocated = .false.
    end if
    if (object%f_allocated) then
      deallocate (object%f)
      object%f_allocated = .false.
    end if
  end subroutine kinematics_final

```

```
end subroutine kinematics_final
```

Allocate the structure-function chain instance, initialize it as a copy of the `sf_chain` template, and prepare it for evaluation.

The `sf_chain` remains a target because the (constant) beam momenta are taken from there.

```
<Processes: kinematics: TBP>+≡
```

```
  procedure :: init_sf_chain => kinematics_init_sf_chain
```

```
<Processes: procedures>+≡
```

```
  subroutine kinematics_init_sf_chain (k, core, sf_chain, config, tmp)
    class(kinematics_t), intent(inout) :: k
    class(prc_core_t), intent(in) :: core
    type(sf_chain_t), intent(in), target :: sf_chain
    type(process_beam_config_t), intent(in) :: config
    class(workspace_t), intent(inout), allocatable :: tmp
    integer :: n_strfun, n_channel
    integer :: i, c
    k%n_in = beam_data_get_n_in (config%data)
    n_strfun = config%n_strfun
    n_channel = config%n_channel
    allocate (k%sf_chain)
    k%sf_chain_allocated = .true.
    call core%init_sf_chain (k%sf_chain, sf_chain, n_channel, tmp)
    if (n_strfun /= 0) then
      do c = 1, n_channel
        call k%sf_chain%set_channel (c, config%sf_channel(c))
      end do
    end if
    call k%sf_chain%link_interactions ()
    call k%sf_chain%exchange_mask ()
    call k%sf_chain%init_evaluators ()
  end subroutine kinematics_init_sf_chain
```

Allocate and initialize the phase-space part and the array of Jacobian factors.

```
<Processes: kinematics: TBP>+≡
```

```
  procedure :: init_phs => kinematics_init_phs
```

```
<Processes: procedures>+≡
```

```
  subroutine kinematics_init_phs (k, config)
    class(kinematics_t), intent(inout) :: k
    class(phs_config_t), intent(in), target :: config
    k%n_channel = config%get_n_channel ()
    call config%allocate_instance (k%phs)
    call k%phs%init (config)
    k%phs_allocated = .true.
    allocate (k%f (k%n_channel))
    k%f = 0
    k%f_allocated = .true.
  end subroutine kinematics_init_phs
```



Initialize the kinematics in form of simple pointers. In essence, this is a shallow copy, but we have to set the flags correctly to indicate this fact.

```

(Processes: kinematics: TBP)+≡
  procedure :: init_ptr => kinematics_init_ptr

(Processes: procedures)+≡
  subroutine kinematics_init_ptr (k, k_in)
    class(kinematics_t), intent(out) :: k
    type(kinematics_t), intent(in) :: k_in
    k%n_in = k_in%n_in
    k%n_channel = k_in%n_channel
    k%sf_chain => k_in%sf_chain
    k%phs => k_in%phs
    k%f => k_in%f
  end subroutine kinematics_init_ptr

```

Generate complete kinematics, given a phase-space channel and a MC parameter set. The main result is the momentum array *p*, but we also fill the momentum entries in the structure-function chain and the Jacobian-factor array *f*.

```

(Processes: kinematics: TBP)+≡
  procedure :: compute => kinematics_compute

(Processes: procedures)+≡
  subroutine kinematics_compute (k, mci_work, phs_channel, p, success)
    class(kinematics_t), intent(inout) :: k
    type(mci_work_t), intent(in) :: mci_work
    integer, intent(in) :: phs_channel
    type(vector4_t), dimension(:), intent(out) :: p
    logical, intent(out) :: success
    integer :: c, c_sf
    integer :: sf_channel
    k%selected_channel = phs_channel
    sf_channel = k%phs%config%get_sf_channel (phs_channel)
    call k%sf_chain%compute_kinematics (sf_channel, mci_work%get_x_strfun ())
    call k%sf_chain%get_out_momenta (p(1:k%n_in))
    call k%phs%set_incoming_momenta (p(1:k%n_in))
    call k%phs%compute_flux ()
    call k%phs%select_channel (phs_channel)
    call k%phs%evaluate (phs_channel, mci_work%get_x_process ())
    if (k%phs%q_defined) then
      call k%phs%get_outgoing_momenta (p(k%n_in+1:))
      do c = 1, k%n_channel
        c_sf = k%phs%config%get_sf_channel (c)
        k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
      end do
      k%phs_factor = k%phs%get_overall_factor ()
      success = .true.
    else
      k%phs_factor = 0
      success = .false.
    end if
  end subroutine kinematics_compute

```

Just fetch the outgoing momenta of the `sf_chain` subobject, which become the incoming (seed) momenta of the hard interaction.

This is a stripped down-version of the above which we use when recovering kinematics. Momenta are known, but no MC parameters yet.

(We do not use the `get_out_momenta` method of the chain, since this relies on the structure-function interactions, which are not necessary filled here. We do rely on the momenta of the last evaluator in the chain, however.)

```

(Processes: kinematics: TBP)+≡
  procedure :: get_incoming_momenta => kinematics_get_incoming_momenta

(Processes: procedures)+≡
  subroutine kinematics_get_incoming_momenta (k, p)
    class(kinematics_t), intent(in) :: k
    type(vector4_t), dimension(:), intent(out) :: p
    type(interaction_t), pointer :: int
    integer :: i
    int => k%sf_chain%get_out_int_ptr ()
    do i = 1, k%n_in
      p(i) = interaction_get_momentum (int, k%sf_chain%get_out_i (i))
    end do
  end subroutine kinematics_get_incoming_momenta

```

This inverts the remainder of the above `compute` method. We know the momenta and recover the rest, as far as needed. If we select a channel, we can complete the inversion and reconstruct the MC parameter set.

```

(Processes: kinematics: TBP)+≡
  procedure :: recover_mcpair => kinematics_recover_mcpair

(Processes: procedures)+≡
  subroutine kinematics_recover_mcpair (k, mci_work, phs_channel, p)
    class(kinematics_t), intent(inout) :: k
    type(mci_work_t), intent(inout) :: mci_work
    integer, intent(in) :: phs_channel
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: c, c_sf
    real(default), dimension(:), allocatable :: x_sf, x_phs
    c = phs_channel
    c_sf = k%phs%config%get_sf_channel (c)
    k%selected_channel = c
    call k%sf_chain%recover_kinematics (c_sf)
    call k%phs%set_incoming_momenta (p(1:k%n_in))
    call k%phs%compute_flux ()
    call k%phs%set_outgoing_momenta (p(k%n_in+1:))
    call k%phs%inverse ()
    do c = 1, k%n_channel
      c_sf = k%phs%config%get_sf_channel (c)
      k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
    end do
    k%phs_factor = k%phs%get_overall_factor ()
    c = phs_channel
    c_sf = k%phs%config%get_sf_channel (c)
    allocate (x_sf (k%sf_chain%config%get_n_par ()))
    allocate (x_phs (k%phs%config%get_n_par ()))
    call k%phs%select_channel (c)

```

```

    call k%sf_chain%get_mcpair (c_sf, x_sf)
    call k%phs%get_mcpair (c, x_phs)
    call mci_work%set_x_strfun (x_sf)
    call mci_work%set_x_process (x_phs)
end subroutine kinematics_recover_mcpair

```

Retrieve the MC input parameter array for a specific channel. We assume that the kinematics is complete, so this is known for all channels.

```

<Processes: kinematics: TBP>+≡
  procedure :: get_mcpair => kinematics_get_mcpair

<Processes: procedures>+≡
  subroutine kinematics_get_mcpair (k, phs_channel, r)
    class(kinematics_t), intent(in) :: k
    integer, intent(in) :: phs_channel
    real(default), dimension(:), intent(out) :: r
    integer :: sf_channel, n_par_sf, n_par_phs
    sf_channel = k%phs%config%get_sf_channel (phs_channel)
    n_par_phs = k%phs%config%get_n_par ()
    n_par_sf = k%sf_chain%config%get_n_par ()
    if (n_par_sf > 0) then
      call k%sf_chain%get_mcpair (sf_channel, r(1:n_par_sf))
    end if
    if (n_par_phs > 0) then
      call k%phs%get_mcpair (phs_channel, r(n_par_sf+1:))
    end if
  end subroutine kinematics_get_mcpair

```

Evaluate the structure function chain, assuming that kinematics is known.

The status must be precisely SF\_DONE\_KINEMATICS. We thus avoid evaluating the chain twice via different pointers to the same target.

```

<Processes: kinematics: TBP>+≡
  procedure :: evaluate_sf_chain => kinematics_evaluate_sf_chain

<Processes: procedures>+≡
  subroutine kinematics_evaluate_sf_chain (k, fac_scale)
    class(kinematics_t), intent(inout) :: k
    real(default), intent(in) :: fac_scale
    select case (k%sf_chain%get_status ())
    case (SF_DONE_KINEMATICS)
      call k%sf_chain%evaluate (fac_scale)
    end select
  end subroutine kinematics_evaluate_sf_chain

```

## Process component instance

The actual calculation of a sampling point is done from here.

The `config` pointer accesses the corresponding configuration in the `process` object.

The `active` flag indicates that we are currently computing this component, together with all other components that share the same MC parameter set.

Inactive components are using a different MC parameter set and are not in use for this sampling point.

The `k_seed` subobject contains the kinematics (structure-function chain, phase space, etc.) that implements the ‘seed’ configuration of momenta. This version of the process kinematics is accessed by the MCI setup.

`p_seed` is the array of momenta that we compute from the MC input parameters, via the `k_seed` subobject. Depending on the process variant, these may or may not coincide with the momenta that enter the process terms associated to this component.

The `tmp` object can be used for storing intermediate results. Its precise type and contents depend on the process variant.

```

(Processes: types)+≡
  type :: component_instance_t
    type(process_component_t), pointer :: config => null ()
    logical :: active = .false.
    type(kinematics_t) :: k_seed
    type(vector4_t), dimension(:), allocatable :: p_seed
    logical :: sqme_known = .false.
    real(default) :: sqme = 0
    class(workspace_t), allocatable :: tmp
  contains
    (Processes: component instance: TBP)
  end type component_instance_t

```

In the header, fetch the component index from the configuration record. `process_component_t` configuration block.

We write the `sf_chain` subobject only upon request, since its instances appear elsewhere.

```

(Processes: component instance: TBP)≡
  procedure :: write => component_instance_write

(Processes: procedures)+≡
  subroutine component_instance_write (object, unit)
    class(component_instance_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i, c
    u = output_unit (unit)
    if (object%active) then
      if (associated (object%config)) then
        write (u, "(1x,A,I0)") "Component #", object%config%index
      else
        write (u, "(1x,A)") "Component [undefined]"
      end if
    else
      write (u, "(1x,A,I0,A)") "Component #", object%config%index, &
        " [inactive]"
    end if
    if (allocated (object%p_seed)) then
      write (u, "(1x,A)") "Seed momenta:"
      do i = 1, size (object%p_seed)
        call vector4_write (object%p_seed(i), u)
      end do
    end if
  end subroutine

```

```

write (u, "(1x,A)") "Squared matrix element:"
if (object%sqme_known) then
  write (u, "(2x,ES19.12)") object%sqme
else
  write (u, "(2x,A)") "[undefined]"
end if
call object%k_seed%write (u)
if (allocated (object%tmp)) then
  call write_separator (u)
  call object%tmp%write (u)
end if
end subroutine component_instance_write

```

Finalizer

```

<Processes: component instance: TBP>+≡
  procedure :: final => component_instance_final

<Processes: procedures>+≡
  subroutine component_instance_final (object)
    class(component_instance_t), intent(inout) :: object
    call object%k_seed%final ()
  end subroutine component_instance_final

```

Initialize: associate the configuration pointer. Also initialize the process workspace, if there is anything to do. This initialization is a matter of the process core.

```

<Processes: component instance: TBP>+≡
  procedure :: init => component_instance_init

<Processes: procedures>+≡
  subroutine component_instance_init (component, config)
    class(component_instance_t), intent(out) :: component
    type(process_component_t), intent(in), target :: config
    integer :: n_in, n_tot
    component%config => config
    associate (core => component%config%core)
      n_in = core%data%n_in
      n_tot = n_in + core%data%n_out
      allocate (component%p_seed (n_tot))
      call core%allocate_workspace (component%tmp)
    end associate
  end subroutine component_instance_init

```

Initialize the seed-kinematics configuration. All subobjects are allocated explicitly.

```

<Processes: component instance: TBP>+≡
  procedure :: setup_kinematics => component_instance_setup_kinematics

<Processes: procedures>+≡
  subroutine component_instance_setup_kinematics (component, sf_chain, config)
    class(component_instance_t), intent(inout) :: component
    type(sf_chain_t), intent(in), target :: sf_chain
    type(process_beam_config_t), intent(in) :: config
    call component%k_seed%init_sf_chain &

```

```

      (component%config%core, sf_chain, config, component%tmp)
      call component%k_seed%init_phs (component%config%phs_config)
    end subroutine component_instance_setup_kinematics

```

Setup seed kinematics, starting from the MC parameter set given as argument. As a result, the `k_seed` kinematics object is evaluated (except for the structure-function matrix-element evaluation, which we postpone until we know the factorization scale), and we have a valid `p_seed` momentum array.

```

<Processes: component instance: TBP>+≡
  procedure :: compute_seed_kinematics => &
    component_instance_compute_seed_kinematics

<Processes: procedures>+≡
  subroutine component_instance_compute_seed_kinematics &
    (component, mci_work, phs_channel, success)
    class(component_instance_t), intent(inout), target :: component
    type(mci_work_t), intent(in) :: mci_work
    integer, intent(in) :: phs_channel
    logical, intent(out) :: success
    call component%k_seed%compute &
      (mci_work, phs_channel, component%p_seed, success)
  end subroutine component_instance_compute_seed_kinematics

```

Inverse: recover missing parts of the kinematics, given a complete set of seed momenta. Select a channel and reconstruct the MC parameter set.

```

<Processes: component instance: TBP>+≡
  procedure :: recover_mcpair => component_instance_recover_mcpair

<Processes: procedures>+≡
  subroutine component_instance_recover_mcpair (component, mci_work, phs_channel)
    class(component_instance_t), intent(inout), target :: component
    type(mci_work_t), intent(inout) :: mci_work
    integer, intent(in) :: phs_channel
    call component%k_seed%recover_mcpair &
      (mci_work, phs_channel, component%p_seed)
  end subroutine component_instance_recover_mcpair

```

Compute the momenta in the hard interactions, one for each term that constitutes this process component. In simple cases this amounts to just copying momenta. In more advanced cases, we may generate distinct sets of momenta from the seed kinematics.

The interactions in the term instances are accessed individually. We may choose to calculate all terms at once together with the seed kinematics, use `component%tmp` for storage, and just fill the interactions here.

```

<Processes: component instance: TBP>+≡
  procedure :: compute_hard_kinematics => &
    component_instance_compute_hard_kinematics

<Processes: procedures>+≡
  subroutine component_instance_compute_hard_kinematics &
    (component, term, skip_term)
    class(component_instance_t), intent(inout) :: component
    type(term_instance_t), dimension(:), intent(inout) :: term

```

```

integer, intent(in), optional :: skip_term
integer :: j, i
associate (core => component%config%core)
  associate (i_term => component%config%i_term)
    do j = 1, size (i_term)
      i = i_term(j)
      if (present (skip_term)) then
        if (i == skip_term) cycle
      end if
      call core%compute_hard_kinematics &
        (component%p_seed, i, term(i)%int_hard, component%tmp)
    end do
  end associate
end associate
end subroutine component_instance_compute_hard_kinematics

```

Here, we invert this. We fetch the incoming momenta which reside in the appropriate `sf_chain` object, stored within the `k_seed` subobject. On the other hand, we have the outgoing momenta of the effective interaction. We rely on the process core to compute the remaining seed momenta and to fill the momenta within the hard interaction. (The latter is trivial if hard and effective interaction coincide.)

After this is done, the incoming momenta in the trace evaluator that corresponds to the hard (effective) interaction, are still left undefined. We remedy this by calling `receive_kinematics` once.

```

(Processes: component instance: TBP)+≡
  procedure :: recover_seed_kinematics => &
    component_instance_recover_seed_kinematics

(Processes: procedures)+≡
  subroutine component_instance_recover_seed_kinematics (component, term)
    class(component_instance_t), intent(inout) :: component
    type(term_instance_t), intent(inout) :: term
    integer :: n_in
    n_in = component%k_seed%n_in
    call component%k_seed%get_incoming_momenta (component%p_seed(1:n_in))
    associate (core => component%config%core)
      call core%recover_kinematics &
        (component%p_seed, term%int_hard, term%isolated%int_eff, &
         component%tmp)
      call term%isolated%receive_kinematics ()
    end associate
  end subroutine component_instance_recover_seed_kinematics

```

Evaluate the trace of the transition matrix, convoluted with the initial state, and summed over all terms. The trace evaluators of the individual terms have only a single matrix element. We implicitly drop the imaginary part of the terms, which should be zero anyway.

```

(Processes: component instance: TBP)+≡
  procedure :: evaluate_sqme => component_instance_evaluate_sqme

(Processes: procedures)+≡
  subroutine component_instance_evaluate_sqme (component, term)

```

```

class(component_instance_t), intent(inout) :: component
type(term_instance_t), dimension(:), intent(in), target :: term
type(interaction_t), pointer :: int
real(default) :: sqme
integer :: j, i
component%sqme = 0
associate (i_term => component%config%i_term)
  do j = 1, size (i_term)
    i = i_term(j)
    if (term(i)%passed) then
      int => evaluator_get_int_ptr (term(i)%connected%trace)
      sqme = interaction_get_matrix_element (int, 1)
      component%sqme = component%sqme + sqme * term(i)%weight
    end if
  end do
end associate
component%sqme_known = .true.
end subroutine component_instance_evaluate_sqme

```

## Term instance

A `term_instance_t` object contains all data that describe a term. Each process component consists of one or more distinct terms which may differ in kinematics, but whose squared transition matrices have to be added pointwise.

The `active` flag is set when this term is connected to an active process component. Inactive terms are skipped for kinematics and evaluation.

The `k_term` object is the instance of the kinematics setup (structure-function chain, phase space, etc.) that applies specifically to this term. In ordinary cases, it consists of straight pointers to the seed kinematics.

The `amp` array stores the amplitude values when we get them from evaluating the associated matrix-element code.

The `int_hard` interaction describes the elementary hard process. It receives the momenta and the amplitude entries for each sampling point.

The `isolated` object holds the effective parton state for the elementary interaction. The amplitude entries are computed from `int_hard`.

The `connected` evaluator set convolutes this scattering matrix with the beam (and possibly structure-function) density matrix.

The `checked` flag is set once we have applied cuts on this term. The result of this is stored in the `passed` flag. Once the term has passed cuts, we calculate the various scale and weight expressions.

*(Processes: types)+≡*

```

type :: term_instance_t
  type(process_term_t), pointer :: config => null ()
  logical :: active = .false.
  type(kinematics_t) :: k_term
  complex(default), dimension(:), allocatable :: amp
  type(interaction_t) :: int_hard
  type(isolated_state_t) :: isolated
  type(connected_state_t) :: connected
  logical :: checked = .false.
  logical :: passed = .false.

```



```

        real(default) :: scale = 0
        real(default) :: fac_scale = 0
        real(default) :: ren_scale = 0
        real(default) :: weight = 1
    contains
        <Processes: term instance: TBP>
    end type term_instance_t

<Processes: term instance: TBP>≡
    procedure :: write => term_instance_write

<Processes: procedures>+≡
    subroutine term_instance_write (term, unit, show_eff_state)
        class(term_instance_t), intent(in) :: term
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: show_eff_state
        integer :: u
        logical :: state
        u = output_unit (unit)
        state = .true.; if (present (show_eff_state)) state = show_eff_state
        if (term%active) then
            if (associated (term%config)) then
                write (u, "(1x,A,IO,A,IO,A)") "Term #", term%config%i_term, &
                    " (component #", term%config%i_component, ")"
            else
                write (u, "(1x,A)") "Term [undefined]"
            end if
        else
            write (u, "(1x,A,IO,A)") "Term #", term%config%i_term, &
                " [inactive]"
        end if
        if (term%checked) then
            write (u, "(3x,A,L1)") "passed cuts" = ", term%passed
        end if
        if (term%passed) then
            write (u, "(3x,A,ES19.12)") "overall scale" = ", term%scale
            write (u, "(3x,A,ES19.12)") "factorization scale" = ", term%fac_scale
            write (u, "(3x,A,ES19.12)") "renormalization scale" = ", term%ren_scale
            write (u, "(3x,A,ES19.12)") "reweighting factor" = ", term%weight
        end if
        call term%k_term%write (u)
        call write_separator (u)
        write (u, "(1x,A)") "Amplitude (transition matrix of the &
            &hard interaction):"
        call write_separator (u)
        call interaction_write (term%int_hard, u)
        if (state .and. term%isolated%has_trace) then
            call write_separator (u)
            write (u, "(1x,A)") "Evaluators for the hard interaction:"
            call term%isolated%write (u)
        end if
        if (state .and. term%connected%has_trace) then
            call write_separator (u)
            write (u, "(1x,A)") "Evaluators for the connected process:"

```

```

        call term%connected%write (u)
    end if
end subroutine term_instance_write

```

The interactions and evaluators must be finalized.

```

<Processes: term instance: TBP>+=
    procedure :: final => term_instance_final

<Processes: procedures>+=
    subroutine term_instance_final (term)
        class(term_instance_t), intent(inout) :: term
        call term%k_term%final ()
        call term%connected%final ()
        call term%isolated%final ()
        call interaction_final (term%int_hard)
    end subroutine term_instance_final

```

For initialization, we make use of defined assignment for the `interaction_t` type. This creates a deep copy.

The hard interaction (incoming momenta) is linked to the structure function instance. In the isolated state, we either set pointers to both, or we create modified copies (`rearrange`) as effective structure-function chain and interaction, respectively.

Finally, we set up the `subevt` component that will be used for evaluating observables, collecting particles from the trace evaluator in the effective connected state. Their quantum numbers must be determined by following back source links and set explicitly, since they are already eliminated in that trace.

```

<Processes: term instance: TBP>+=
    procedure :: init => term_instance_init

<Processes: procedures>+=
    subroutine term_instance_init (term, &
        config, k_seed, beam_config, core, process_var_list)
        class(term_instance_t), intent(out), target :: term
        type(process_term_t), intent(in), target :: config
        type(kinematics_t), intent(in) :: k_seed
        type(process_beam_config_t), intent(in) :: beam_config
        type(interaction_t), pointer :: sf_chain_int, trace_int, src_int
        class(prc_core_t), intent(in) :: core
        type(var_list_t), intent(in), target :: process_var_list
        type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
        type(state_matrix_t), pointer :: state_matrix
        type(flavor_t), dimension(:), allocatable :: flv_int, flv_src, f_in, f_out
        integer :: n_in, n_vir, n_out, n_tot
        integer :: i, j
        term%config => config
        if (config%rearrange) then
            ! rearrangement of seed to hard kinematics not implemented yet
            ! allocate k_term distinct from k_seed as needed.
        else
            ! here, k_term trivially accesses k_seed via pointers
            call term%k_term%init_ptr (k_seed)
        end if
    end subroutine term_instance_init

```

```

allocate (term%amp (config%n_allowed))
term%int_hard = config%int
sf_chain_int => term%k_term%sf_chain%get_out_int_ptr ()
n_in = interaction_get_n_in (term%int_hard)
do j = 1, n_in
    i = term%k_term%sf_chain%get_out_i (j)
    call interaction_set_source_link (term%int_hard, j, sf_chain_int, i)
end do
if (config%rearrange) then
    ! rearrangement hard to effective kinematics not implemented yet
    ! should use term%config%int_eff as template
    ! allocate distinct sf_chain in term%connected as needed
else
    ! here, int_hard and sf_chain are trivially accessed via pointers
    call term%isolated%init (term%k_term%sf_chain, term%int_hard)
end if
!!! Catching Intel 13.1 problem with auto-allocate
allocate (mask_in (n_in))
mask_in = term%k_term%sf_chain%get_out_mask ()
call term%isolated%setup_square_trace (core, mask_in, term%config%col)
call term%connected%setup_connected_trace (term%isolated)
associate (int_eff => term%isolated%int_eff)
    state_matrix => interaction_get_state_matrix_ptr (int_eff)
    n_tot = interaction_get_n_tot (int_eff)
    allocate (flv_int (n_tot))
    flv_int = quantum_numbers_get_flavor &
        (state_matrix_get_quantum_numbers (state_matrix, 1))
    allocate (f_in (n_in))
    f_in = flv_int(1:n_in)
    deallocate (flv_int)
end associate
trace_int => evaluator_get_int_ptr (term%connected%trace)
n_in = interaction_get_n_in (trace_int)
n_vir = interaction_get_n_vir (trace_int)
n_out = interaction_get_n_out (trace_int)
allocate (f_out (n_out))
do j = 1, n_out
    call interaction_find_source (trace_int, n_in + n_vir + j, src_int, i)
    if (associated (src_int)) then
        state_matrix => interaction_get_state_matrix_ptr (src_int)
        allocate (flv_src (interaction_get_n_tot (src_int)))
        flv_src = quantum_numbers_get_flavor &
            (state_matrix_get_quantum_numbers (state_matrix, 1))
        f_out(j) = flv_src(i)
        deallocate (flv_src)
    end if
end do
call term%connected%setup_subevt (term%isolated%sf_chain_eff, &
    beam_config%data%flv, f_in, f_out)
call term%connected%setup_var_list (process_var_list, beam_config%data)
end subroutine term_instance_init

```

For initializing the expressions, we need the local variable list and the parse trees.

```

<Processes: term instance: TBP>+≡
  procedure :: setup_expressions => term_instance_setup_expressions

<Processes: procedures>+≡
  subroutine term_instance_setup_expressions (term, meta, config)
    class(term_instance_t), intent(inout), target :: term
    type(process_metadata_t), intent(in), target :: meta
    type(process_config_data_t), intent(in) :: config
    call term%connected%setup_expressions ( &
      config%pn_cuts, &
      config%pn_scale, &
      config%pn_fac_scale, &
      config%pn_ren_scale, &
      config%pn_weight)
  end subroutine term_instance_setup_expressions

```

Prepare the extra evaluators that we need for processing events.

```

<Processes: term instance: TBP>+≡
  procedure :: setup_event_data => term_instance_setup_event_data

<Processes: procedures>+≡
  subroutine term_instance_setup_event_data (term, core, model)
    class(term_instance_t), intent(inout), target :: term
    class(prc_core_t), intent(in) :: core
    type(model_t), intent(in), target :: model
    integer :: n_in
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
    n_in = interaction_get_n_in (term%int_hard)
    allocate (mask_in (n_in))
    mask_in = term%k_term%sf_chain%get_out_mask ()
    call term%isolated%setup_square_matrix (core, model, mask_in, &
      term%config%col)
    call term%isolated%setup_square_flows (core, model, mask_in)
    call term%connected%setup_connected_matrix (term%isolated)
    call term%connected%setup_connected_flows (term%isolated)
  end subroutine term_instance_setup_event_data

```

Reset the term instance: clear the parton-state expressions and deactivate.

```

<Processes: term instance: TBP>+≡
  procedure :: reset => term_instance_reset

<Processes: procedures>+≡
  subroutine term_instance_reset (term)
    class(term_instance_t), intent(inout) :: term
    call term%connected%reset_expressions ()
    term%active = .false.
  end subroutine term_instance_reset

```

Complete the kinematics computation for the effective parton states.

We assume that the `compute_hard_kinematics` method of the process component instance has already been called, so the `int_hard` contains the correct hard kinematics. The duty of this procedure is first to compute the effective kinematics and store this in the `int_eff` effective interaction inside the

isolated parton state. The effective kinematics may differ from the kinematics in the hard interaction. It may involve parton recombination or parton splitting. The `rearrange_partons` method is responsible for this part.

We may also call a method to compute the effective structure-function chain at this point. This is not implemented yet.

In the simple case that no rearrangement is necessary, as indicated by the `rearrange` flag, the effective interaction is a pointer to the hard interaction, and we can skip the rearrangement method. Similarly for the effective structure-function chain.

The final step of kinematics setup is to transfer the effective kinematics to the evaluators and to the `subevt`.

```

(Processes: term instance: TBP)+≡
  procedure :: compute_eff_kinematics => &
    term_instance_compute_eff_kinematics

(Processes: procedures)+≡
  subroutine term_instance_compute_eff_kinematics (term, component)
    class(term_instance_t), intent(inout) :: term
    type(component_instance_t), dimension(:), intent(inout) :: component
    integer :: i_component, i_term
    term%checked = .false.
    term%passed = .false.
    if (term%config%rearrange) then
      ! should evaluate k_term first if allocated separately, not impl. yet
      i_component = term%config%i_component
      i_term = term%config%i_term
      associate (core => component(i_component)%config%core)
        call core%compute_eff_kinematics &
          (i_term, term%int_hard, term%isolated%int_eff, &
            component(i_component)%tmp)
      end associate
    end if
    call term%isolated%receive_kinematics ()
    call term%connected%receive_kinematics ()
  end subroutine term_instance_compute_eff_kinematics

```

Inverse. Reconstruct the connected state from the momenta in the trace evaluator (which we assume to be set), then reconstruct the isolated state as far as possible. The second part finalizes the momentum configuration, using the incoming seed momenta

```

(Processes: term instance: TBP)+≡
  procedure :: recover_hard_kinematics => &
    term_instance_recover_hard_kinematics

(Processes: procedures)+≡
  subroutine term_instance_recover_hard_kinematics (term, component)
    class(term_instance_t), intent(inout) :: term
    type(component_instance_t), dimension(:), intent(inout) :: component
    integer :: i_component, i_term
    term%checked = .false.
    term%passed = .false.
    call term%connected%send_kinematics ()
    call term%isolated%send_kinematics ()
  end subroutine term_instance_recover_hard_kinematics

```

```
end subroutine term_instance_recover_hard_kinematics
```

Check the term whether it passes cuts and, if successful, evaluate scales and weights. The factorization scale is also given to the term kinematics, enabling structure-function evaluation.

```
<Processes: term instance: TBP>+≡
  procedure :: evaluate_expressions => &
    term_instance_evaluate_expressions

<Processes: procedures>+≡
  subroutine term_instance_evaluate_expressions (term)
    class(term_instance_t), intent(inout) :: term
    call term%connected%evaluate_expressions (term%passed, &
      term%scale, term%fac_scale, term%ren_scale, term%weight)
    term%checked = .true.
  end subroutine term_instance_evaluate_expressions
```

Evaluate the trace: first evaluate the hard interaction, then the trace evaluator. We use the `evaluate_interaction` method of the process component which generated this term. The `subevt` and cut expressions are not yet filled.

The `component` argument is `intent(inout)` because the `compute_amplitude` method may modify the `tmp` workspace object.

```
<Processes: term instance: TBP>+≡
  procedure :: evaluate_interaction => term_instance_evaluate_interaction

<Processes: procedures>+≡
  subroutine term_instance_evaluate_interaction (term, component)
    class(term_instance_t), intent(inout) :: term
    type(component_instance_t), dimension(:), intent(inout) :: component
    integer :: i_component, i_term, i
    i_component = term%config%i_component
    i_term = term%config%i_term
    associate (core => component(i_component)%config%core)
      do i = 1, term%config%n_allowed
        term%amp(i) = core%compute_amplitude (i_term, &
          interaction_get_momenta (term%int_hard), &
          term%config%flv(i), term%config%hel(i), term%config%col(i), &
          term%fac_scale, term%ren_scale, &
          component(i_component)%tmp)
      end do
      call interaction_set_matrix_element (term%int_hard, term%amp)
    end associate
  end subroutine term_instance_evaluate_interaction
```

Evaluate the trace. First evaluate the structure-function chain (i.e., the density matrix of the incoming partons). Do this twice, in case the `sf-chain` instances within `k_term` and `isolated` differ. Next, evaluate the hard interaction, then compute the convolution with the initial state.

```
<Processes: term instance: TBP>+≡
  procedure :: evaluate_trace => term_instance_evaluate_trace
```

```

<Processes: procedures>+≡
subroutine term_instance_evaluate_trace (term)
  class(term_instance_t), intent(inout) :: term
  call term%k_term%evaluate_sf_chain (term%fac_scale)
  call term%isolated%evaluate_sf_chain (term%fac_scale)
  call term%isolated%evaluate_trace ()
  call term%connected%evaluate_trace ()
end subroutine term_instance_evaluate_trace

```

Evaluate the extra data that we need for processing the object as a physical event.

```

<Processes: term instance: TBP>+≡
  procedure :: evaluate_event_data => term_instance_evaluate_event_data

<Processes: procedures>+≡
subroutine term_instance_evaluate_event_data (term)
  class(term_instance_t), intent(inout) :: term
  call term%isolated%evaluate_event_data ()
  call term%connected%evaluate_event_data ()
end subroutine term_instance_evaluate_event_data

```

Return data that might be useful for external processing. The factorization scale:

```

<Processes: term instance: TBP>+≡
  procedure :: get_fac_scale => term_instance_get_fac_scale

<Processes: procedures>+≡
function term_instance_get_fac_scale (term) result (fac_scale)
  class(term_instance_t), intent(in) :: term
  real(default) :: fac_scale
  fac_scale = term%fac_scale
end function term_instance_get_fac_scale

```

We take the strong coupling from the process core. The value is calculated when a new event is requested, so we should call it only after the event has been evaluated. If it is not available there (a negative number is returned), we take the value stored in the term configuration, which should be determined by the model. If the model does not provide a value, the result is zero.

```

<Processes: term instance: TBP>+≡
  procedure :: get_alpha_s => term_instance_get_alpha_s

<Processes: procedures>+≡
function term_instance_get_alpha_s (term, component) result (alpha_s)
  class(term_instance_t), intent(in) :: term
  type(component_instance_t), dimension(:), intent(in) :: component
  real(default) :: alpha_s
  integer :: i_component
  i_component = term%config%i_component
  associate (core => component(i_component)%config%core)
    alpha_s = core%get_alpha_s (component(i_component)%tmp)
  end associate
  if (alpha_s < 0) alpha_s = term%config%alpha_s
end function term_instance_get_alpha_s

```

## MC parameter set and MCI instance

For each process component that is associated with a multi-channel integration (MCI) object, the `mci_work_t` object contains the currently active parameter set. It also holds the implementation of the `mci_instance_t` that the integrator needs for doing its work.

```

<Processes: types>+≡
  type :: mci_work_t
    type(process_mci_entry_t), pointer :: config => null ()
    real(default), dimension(:), allocatable :: x
    class(mci_instance_t), pointer :: mci => null ()
    contains
    <Processes: mci work: TBP>
  end type mci_work_t

```

First write configuration data, then the current values.

```

<Processes: mci work: TBP>≡
  procedure :: write => mci_work_write

<Processes: procedures>+≡
  subroutine mci_work_write (mci_work, unit)
    class(mci_work_t), intent(in) :: mci_work
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(1x,A,I0,A)") "Active MCI instance #", &
      mci_work%config%i_mci, " ="
    write (u, "(2x)", advance="no")
    do i = 1, mci_work%config%n_par
      write (u, "(1x,F7.5)", advance="no") mci_work%x(i)
      if (i == mci_work%config%n_par_sf) &
        write (u, "(1x,'|')", advance="no")
    end do
    write (u, *)
    if (associated (mci_work%mci)) then
      call mci_work%mci%write (u)
    end if
  end subroutine mci_work_write

```

The `mci` component may require finalization.

```

<Processes: mci work: TBP>+≡
  procedure :: final => mci_work_final

<Processes: procedures>+≡
  subroutine mci_work_final (mci_work)
    class(mci_work_t), intent(inout) :: mci_work
    if (associated (mci_work%mci)) then
      call mci_work%mci%final ()
      deallocate (mci_work%mci)
    end if
  end subroutine mci_work_final

```



Initialize with the maximum length that we will need. Contents are not initialized.

The integrator inside the `mci_entry` object is responsible for allocating and initializing its own instance, which is referred to by a pointer in the `mci_work` object.

```

<Processes: mci work: TBP>+≡
  procedure :: init => mci_work_init

<Processes: procedures>+≡
  subroutine mci_work_init (mci_work, mci_entry)
    class(mci_work_t), intent(out) :: mci_work
    type(process_mci_entry_t), intent(in), target :: mci_entry
    mci_work%config => mci_entry
    allocate (mci_work%x (mci_entry%n_par))
    if (allocated (mci_entry%mci)) then
      call mci_entry%mci%allocate_instance (mci_work%mci)
      call mci_work%mci%init (mci_entry%mci)
    end if
  end subroutine mci_work_init

```

Set parameters explicitly, either all at once, or separately for the structure-function and process parts.

```

<Processes: mci work: TBP>+≡
  procedure :: set => mci_work_set
  procedure :: set_x_strfun => mci_work_set_x_strfun
  procedure :: set_x_process => mci_work_set_x_process

<Processes: procedures>+≡
  subroutine mci_work_set (mci_work, x)
    class(mci_work_t), intent(inout) :: mci_work
    real(default), dimension(:), intent(in) :: x
    mci_work%x = x
  end subroutine mci_work_set

  subroutine mci_work_set_x_strfun (mci_work, x)
    class(mci_work_t), intent(inout) :: mci_work
    real(default), dimension(:), intent(in) :: x
    mci_work%x(1 : mci_work%config%n_par_sf) = x
  end subroutine mci_work_set_x_strfun

  subroutine mci_work_set_x_process (mci_work, x)
    class(mci_work_t), intent(inout) :: mci_work
    real(default), dimension(:), intent(in) :: x
    mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par) = x
  end subroutine mci_work_set_x_process

```

Return the array of active components, i.e., those that correspond to the currently selected MC parameter set.

```

<Processes: mci work: TBP>+≡
  procedure :: get_active_components => mci_work_get_active_components

<Processes: procedures>+≡
  function mci_work_get_active_components (mci_work) result (i_component)
    class(mci_work_t), intent(in) :: mci_work

```

```

integer, dimension(:), allocatable :: i_component
allocate (i_component (size (mci_work%config%i_component)))
i_component = mci_work%config%i_component
end function mci_work_get_active_components

```

Return the active parameters as a simple array with correct length. Do this separately for the structure-function parameters and the process parameters.

```

(Processes: mci work: TBP)+≡
  procedure :: get_x_strfun => mci_work_get_x_strfun
  procedure :: get_x_process => mci_work_get_x_process

(Processes: procedures)+≡
  function mci_work_get_x_strfun (mci_work) result (x)
    class(mci_work_t), intent(in) :: mci_work
    real(default), dimension(mci_work%config%n_par_sf) :: x
    x = mci_work%x(1 : mci_work%config%n_par_sf)
  end function mci_work_get_x_strfun

  function mci_work_get_x_process (mci_work) result (x)
    class(mci_work_t), intent(in) :: mci_work
    real(default), dimension(mci_work%config%n_par_phs) :: x
    x = mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par)
  end function mci_work_get_x_process

```

## The process instance

A process instance contains all process data that depend on the sampling point and thus change often. In essence, it is an event record at the elementary (parton) level. We do not call it such, to avoid confusion with the actual event records. If decays are involved, the latter are compositions of several elementary processes (i.e., their instances).

We implement the process instance as an extension of the `mci_sampler_t` that we need for computing integrals and generate events.

The base type contains: the `integrand`, the `selected_channel`, the two-dimensional array `x` of parameters, and the one-dimensional array `f` of Jacobians. These subobjects are public and used for communicating with the multi-channel integrator.

The `process` pointer accesses the process of which this record is an instance. It is required whenever the calculation needs invariant configuration data, therefore the process should stay in memory for the whole lifetime of its instances.

The `evaluation_status` code is used to check the current status. In particular, failure at various stages is recorded there.

The `sqme` value is the single real number that results from evaluating and tracing the kinematics and matrix elements. This is the number that is handed over to an integration routine.

The `weight` value is the event weight. It is defined when an event has been generated from the process instance, either weighted or unweighted. The value is the `sqme` value times Jacobian weights from the integration, or unity, respectively.

The `i_mci` index chooses a subset of components that are associated with a common parameter set and integrator, i.e., that are added coherently.

The `sf_chain` subobject is a realization of the beam and structure-function configuration in the `process` object. It is not used for calculation directly but serves as the template for the sf-chain instances that are contained in the `component` objects.

The `component` subobjects determine the state of each component.

The `term` subobjects are workspace for evaluating kinematics, matrix elements, cuts etc.

The `mci_work` subobject contains the array of real input parameters (random numbers) that generates the kinematical point. It also contains the workspace for the MC integrators. The active entry of the `mci_work` array is selected by the `i_mci` index above.

```

<Processes: public>+=
  public :: process_instance_t

<Processes: types>+=
  type, extends (mci_sampler_t) :: process_instance_t
    type(process_t), pointer :: process => null ()
    integer :: evaluation_status = STAT_UNDEFINED
    real(default) :: sqme = 0
    real(default) :: weight = 0
    integer :: i_mci = 0
    integer :: selected_channel = 0
    type(sf_chain_t) :: sf_chain
    type(component_instance_t), dimension(:), allocatable :: component
    type(term_instance_t), dimension(:), allocatable :: term
    type(mci_work_t), dimension(:), allocatable :: mci_work
    contains
    <Processes: process instance: TBP>
  end type process_instance_t

<Processes: parameters>=
  integer, parameter :: STAT_UNDEFINED = 0
  integer, parameter :: STAT_INITIAL = 1
  integer, parameter :: STAT_ACTIVATED = 2
  integer, parameter :: STAT_FAILED_KINEMATICS = 3
  integer, parameter :: STAT_SEED_KINEMATICS = 4
  integer, parameter :: STAT_HARD_KINEMATICS = 5
  integer, parameter :: STAT_EFF_KINEMATICS = 6
  integer, parameter :: STAT_FAILED_CUTS = 7
  integer, parameter :: STAT_PASSED_CUTS = 8
  integer, parameter :: STAT_EVALUATED_TRACE = 10
  integer, parameter :: STAT_EVENT_COMPLETE = 11

```

The output routine contains a header with the most relevant information about the process, copied from `process_metadata_write`. We mark the active components by an asterisk.

The next section is the MC parameter input. The following sections are written only if the evaluation status is beyond setting the parameters, or if the `verbose` option is set.

```

<Processes: process instance: TBP>=

```

```

procedure :: write_header => process_instance_write_header
procedure :: write => process_instance_write

(Processes: procedures)+=
subroutine process_instance_write_header (object, unit)
class(process_instance_t), intent(in) :: object
integer, intent(in), optional :: unit
integer :: u, i
u = output_unit (unit)
call write_separator_double (u)
if (associated (object%process)) then
  associate (meta => object%process%meta)
    select case (meta%type)
      case (PRC_UNKNOWN)
        write (u, "(1x,A)") "Process instance [undefined]"
        return
      case (PRC_DECAY)
        write (u, "(1x,A)", advance="no") "Process instance [decay]:"
      case (PRC_SCATTERING)
        write (u, "(1x,A)", advance="no") "Process instance [scattering]:"
      case default
        call msg_bug ("process_instance_write: undefined process type")
    end select
    write (u, "(1x,A,A,A)") "'", char (meta%id), "'"
    write (u, "(3x,A,A,A)") "Run ID = '", char (meta%run_id), "'"
    if (allocated (meta%component_id)) then
      write (u, "(3x,A)") "Process components:"
      do i = 1, size (meta%component_id)
        if (object%component(i)%active) then
          write (u, "(3x,'*')", advance="no")
        else
          write (u, "(4x)", advance="no")
        end if
      end do
      write (u, "(1x,I0,9A)") i, ": '", &
        char (meta%component_id (i)), "' : ", &
        char (meta%component_description (i))
    end do
  end if
end associate
else
  write (u, "(1x,A)") "Process instance [undefined process]"
  return
end if
write (u, "(3x,A)", advance = "no") "status = "
select case (object%evaluation_status)
case (STAT_INITIAL);      write (u, "(A)") "initialized"
case (STAT_ACTIVATED);    write (u, "(A)") "activated"
case (STAT_FAILED_KINEMATICS); write (u, "(A)") "failed kinematics"
case (STAT_SEED_KINEMATICS); write (u, "(A)") "seed kinematics"
case (STAT_HARD_KINEMATICS); write (u, "(A)") "hard kinematics"
case (STAT_EFF_KINEMATICS); write (u, "(A)") "effective kinematics"
case (STAT_FAILED_CUTS);  write (u, "(A)") "failed cuts"
case (STAT_PASSED_CUTS);  write (u, "(A)") "passed cuts"
case (STAT_EVALUATED_TRACE); write (u, "(A)") "evaluated trace"
call write_separator (u)

```

```

        write (u, "(3x,A,ES19.12)") "sqme  = ", object%sqme
    case (STAT_EVENT_COMPLETE);   write (u, "(A)") "event complete"
        call write_separator (u)
        write (u, "(3x,A,ES19.12)") "sqme  = ", object%sqme
        write (u, "(3x,A,ES19.12)") "weight = ", object%weight
    case default;                 write (u, "(A)") "undefined"
end select
if (object%i_mci /= 0) then
    call write_separator (u)
    call object%mc_i_work(object%i_mci)%write (u)
end if
call write_separator_double (u)
end subroutine process_instance_write_header

subroutine process_instance_write (object, unit)
    class(process_instance_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    call object%write_header (u)
    if (object%evaluation_status >= STAT_SEED_KINEMATICS) then
        call object%sf_chain%write (u)
        call write_separator_double (u)
        write (u, "(1x,A)") "Active components:"
        do i = 1, size (object%component)
            if (object%component(i)%active) then
                call write_separator (u)
                call object%component(i)%write (u)
            end if
        end do
    if (object%evaluation_status >= STAT_HARD_KINEMATICS) then
        call write_separator_double (u)
        write (u, "(1x,A)") "Active terms:"
        if (any (object%term%active)) then
            call write_separator (u)
            do i = 1, size (object%term)
                if (object%term(i)%active) then
                    call object%term(i)%write (u, &
                        show_eff_state = &
                        object%evaluation_status >= STAT_EFF_KINEMATICS)
                end if
            end do
        end if
    end if
    call write_separator_double (u)
end if
end subroutine process_instance_write

```

Finalize all subobjects that may contain allocated pointers.

```

<Processes: process instance: TBP>+≡
    procedure :: final => process_instance_final

<Processes: procedures>+≡
    subroutine process_instance_final (instance)

```

```

class(process_instance_t), intent(inout) :: instance
integer :: i
if (allocated (instance%mci_work)) then
    do i = 1, size (instance%mci_work)
        call instance%mci_work(i)%final ()
    end do
end if
call instance%sf_chain%final ()
if (allocated (instance%component)) then
    do i = 1, size (instance%component)
        call instance%component(i)%final ()
    end do
end if
if (allocated (instance%term)) then
    do i = 1, size (instance%term)
        call instance%term(i)%final ()
    end do
end if
instance%evaluation_status = STAT_UNDEFINED
end subroutine process_instance_final

```

Revert the process instance to initial state. We do not deallocate anything, just reset the state index and deactivate all components and terms.

We do not reset the choice of the MCI set `i_mci` unless this is required explicitly.

```

<Processes: process instance: TBP>+≡
    procedure :: reset => process_instance_reset

<Processes: procedures>+≡
    subroutine process_instance_reset (instance, reset_mci)
        class(process_instance_t), intent(inout) :: instance
        logical, intent(in), optional :: reset_mci
        integer :: i
        instance%component%active = .false.
        do i = 1, size (instance%term)
            call instance%term(i)%reset ()
        end do
        instance%term%checked = .false.
        instance%term%passed = .false.
        if (present (reset_mci)) then
            if (reset_mci) instance%i_mci = 0
        end if
        instance%selected_channel = 0
        instance%evaluation_status = STAT_INITIAL
    end subroutine process_instance_reset

```

Activate the components and terms that correspond to a currently selected MCI parameter set.

```

<Processes: process instance: TBP>+≡
    procedure :: activate => process_instance_activate

<Processes: procedures>+≡
    subroutine process_instance_activate (instance)

```

```

class(process_instance_t), intent(inout) :: instance
integer :: i, j
associate (mci_work => instance%mci_work(instance%i_mci))
    instance%component(mci_work%get_active_components ())%active &
        = .true.
    do i = 1, size (instance%component)
        associate (component => instance%component(i))
            do j = 1, size (component%config%i_term)
                instance%term(component%config%i_term(j))%active &
                    = .true.
            end do
        end associate
    end do
end associate
instance%evaluation_status = STAT_ACTIVATED
end subroutine process_instance_activate

```

Initialization connects the instance with a process. All initial information is transferred from the process object. The process object contains templates for the interaction subobjects (beam and term), but no evaluators. The initialization routine creates evaluators for the matrix element trace, other evaluators are left untouched.

The `instance` object must have the `target` attribute (also in any caller) since the initialization routine assigns various pointers to subobject of `instance`.

*(Processes: process instance: TBP)+≡*

```

procedure :: init => process_instance_init

```

*(Processes: procedures)+≡*

```

subroutine process_instance_init (instance, process)
    class(process_instance_t), intent(out), target :: instance
    type(process_t), intent(in), target :: process
    integer :: i, i_component
    instance%process => process
    call instance%setup_sf_chain (process%beam_config)
    allocate (instance%mci_work (process%config%n_mci))
    do i = 1, size (instance%mci_work)
        call instance%mci_work(i)%init (process%mci_entry(i))
    end do
    allocate (instance%component (process%config%n_components))
    do i_component = 1, size (instance%component)
        associate (component => instance%component(i_component))
            call component%init (process%component(i_component))
            call component%setup_kinematics &
                (instance%sf_chain, process%beam_config)
        end associate
    end do
    allocate (instance%term (process%config%n_terms))
    do i = 1, size (instance%term)
        associate (term => instance%term(i))
            i_component = process%term(i)%i_component
            associate (component => instance%component(i_component))
                call term%init (process%term(i), &
                    component%k_seed, &

```

```

        process%beam_config, &
        process%component(i_component)%core, &
        process%meta%var_list)
    call term%setup_expressions (process%meta, process%config)
end associate
end associate
end do
instance%evaluation_status = STAT_INITIAL
end subroutine process_instance_init

```

Subroutine of the initialization above: initialize the beam and structure-function chain template. We establish pointers to the configuration data, so `beam_config` must have a `target` attribute.

The resulting chain is not used directly for calculation. It will acquire instances which are stored in the process-component instance objects.

```

<Processes: process instance: TBP>+≡
    procedure :: setup_sf_chain => process_instance_setup_sf_chain
<Processes: procedures>+≡
    subroutine process_instance_setup_sf_chain (instance, config)
        class(process_instance_t), intent(inout) :: instance
        type(process_beam_config_t), intent(in), target :: config
        integer :: n_strfun
        n_strfun = config%n_strfun
        if (n_strfun /= 0) then
            call instance%sf_chain%init (config%data, config%sf)
        else
            call instance%sf_chain%init (config%data)
        end if
    end subroutine process_instance_setup_sf_chain

```

This initialization routine should be called only for process instances which we intend as a source for physical events. It initializes the evaluators in the parton states of the terms. They describe the (semi-)exclusive transition matrix and the distribution of color flow for the partonic process, convoluted with the beam and structure-function chain.

```

<Processes: process instance: TBP>+≡
    procedure :: setup_event_data => process_instance_setup_event_data
<Processes: procedures>+≡
    subroutine process_instance_setup_event_data (instance)
        class(process_instance_t), intent(inout), target :: instance
        integer :: i, i_component
        do i = 1, size (instance%term)
            associate (term => instance%term(i))
                i_component = term%config%i_component
                associate (component => instance%process%component(i_component))
                    call term%setup_event_data (component%core, &
                        instance%process%config%model)
                end associate
            end associate
        end do
    end subroutine process_instance_setup_event_data

```



Choose a MC parameter set and the corresponding integrator.

The choice persists beyond calls of the `reset` method above. This method is automatically called here.

```

<Processes: process instance: TBP>+≡
  procedure :: choose_mci => process_instance_choose_mci

<Processes: procedures>+≡
  subroutine process_instance_choose_mci (instance, i_mci)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    integer :: i, j
    instance%i_mci = i_mci
    call instance%reset ()
  end subroutine process_instance_choose_mci

```

Explicitly set a MC parameter set. Works only if we are in initial state. We assume that the length of the parameter set is correct.

After setting the parameters, activate the components and terms that correspond to the chosen MC parameter set.

```

<Processes: process instance: TBP>+≡
  procedure :: set_mcpair => process_instance_set_mcpair

<Processes: procedures>+≡
  subroutine process_instance_set_mcpair (instance, x)
    class(process_instance_t), intent(inout) :: instance
    real(default), dimension(:), intent(in) :: x
    integer :: i, j
    if (instance%evaluation_status == STAT_INITIAL) then
      associate (mci_work => instance%mci_work(instance%i_mci))
        call mci_work%set (x)
      end associate
      call instance%activate ()
    else
      ! error?
    end if
  end subroutine process_instance_set_mcpair

```

Explicitly choose MC integration channel. We assume here that the channel count is identical for all active components.

```

<Processes: process instance: TBP>+≡
  procedure :: select_channel => process_instance_select_channel

<Processes: procedures>+≡
  subroutine process_instance_select_channel (instance, channel)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: channel
    instance%selected_channel = channel
  end subroutine process_instance_select_channel

```

First step of process evaluation: set up seed kinematics. That is, for each active process component, compute a momentum array from the MC input parameters.

If `skip_term` is set, we skip the component that accesses this term. We can assume that the associated data have already been recovered, and we are just computing the rest.

```

(Processes: process instance: TBP)+≡
  procedure :: compute_seed_kinematics => &
    process_instance_compute_seed_kinematics

(Processes: procedures)+≡
  subroutine process_instance_compute_seed_kinematics (instance, skip_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in), optional :: skip_term
    integer :: channel, skip_component, i
    logical :: success
    channel = instance%selected_channel
    if (channel == 0) then
      call msg_bug ("Compute seed kinematics: undefined integration channel")
    end if
    if (present (skip_term)) then
      skip_component = instance%term(skip_term)%config%i_component
    else
      skip_component = 0
    end if
    if (instance%evaluation_status >= STAT_ACTIVATED) then
      success = .true.
      do i = 1, size (instance%component)
        if (i == skip_component) cycle
        if (instance%component(i)%active) then
          call instance%component(i)%compute_seed_kinematics &
            (instance%mc_i_work(instance%i_mci), channel, success)
          if (.not. success) exit
        end if
      end do
      if (success) then
        instance%evaluation_status = STAT_SEED_KINEMATICS
      else
        instance%evaluation_status = STAT_FAILED_KINEMATICS
      end if
    else
      ! error?
    end if
  end subroutine process_instance_compute_seed_kinematics

```

Inverse: recover missing parts of the kinematics from the momentum configuration, which we know for a single term and component. Given a channel, reconstruct the MC parameter set.

```

(Processes: process instance: TBP)+≡
  procedure :: recover_mcpair => process_instance_recover_mcpair

(Processes: procedures)+≡
  subroutine process_instance_recover_mcpair (instance, i_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_term
    integer :: channel
    integer :: i

```

```

if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
  channel = instance%selected_channel
  if (channel == 0) then
    call msg_bug ("Recover MC parameters: undefined integration channel")
  end if
  i = instance%term(i_term)%config%i_component
  call instance%component(i)%recover_mcpair &
    (instance%mc_i_work(instance%i_mci), channel)
else
  ! error?
end if
end subroutine process_instance_recover_mcpair

```

Second step of process evaluation: compute all momenta, for all active components, from the seed kinematics.

```

<Processes: process instance: TBP>+≡
  procedure :: compute_hard_kinematics => &
    process_instance_compute_hard_kinematics

<Processes: procedures>+≡
  subroutine process_instance_compute_hard_kinematics (instance, skip_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in), optional :: skip_term
    integer :: i
    if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
      do i = 1, size (instance%component)
        if (instance%component(i)%active) then
          call instance%component(i)% &
            compute_hard_kinematics (instance%term, skip_term)
        end if
      end do
      instance%evaluation_status = STAT_HARD_KINEMATICS
    else
      ! failed kinematics
    end if
  end subroutine process_instance_compute_hard_kinematics

```

Inverse: recover seed kinematics. We know the beam momentum configuration and the outgoing momenta of the effective interaction, for one specific term.

```

<Processes: process instance: TBP>+≡
  procedure :: recover_seed_kinematics => &
    process_instance_recover_seed_kinematics

<Processes: procedures>+≡
  subroutine process_instance_recover_seed_kinematics (instance, i_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_term
    if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
      associate (i_component => instance%term(i_term)%config%i_component)
        call instance%component(i_component)% &
          recover_seed_kinematics (instance%term(i_term))
      end associate
    else
      ! error?
    end if
  end subroutine process_instance_recover_seed_kinematics

```

```

    end if
end subroutine process_instance_recover_seed_kinematics

```

Third step of process evaluation: compute the effective momentum configurations, for all active terms, from the hard kinematics.

```

<Processes: process instance: TBP>+≡
  procedure :: compute_eff_kinematics => &
    process_instance_compute_eff_kinematics

<Processes: procedures>+≡
  subroutine process_instance_compute_eff_kinematics (instance, skip_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in), optional :: skip_term
    integer :: i
    if (instance%evaluation_status >= STAT_HARD_KINEMATICS) then
      do i = 1, size (instance%term)
        if (present (skip_term)) then
          if (i == skip_term) cycle
        end if
        if (instance%term(i)%active) then
          call instance%term(i)% &
            compute_eff_kinematics (instance%component)
        end if
      end do
      instance%evaluation_status = STAT_EFF_KINEMATICS
    else
      ! failed kinematics
    end if
  end subroutine process_instance_compute_eff_kinematics

```

Inverse: recover the hard kinematics from effective kinematics for one term, then compute effective kinematics for the other terms.

```

<Processes: process instance: TBP>+≡
  procedure :: recover_hard_kinematics => &
    process_instance_recover_hard_kinematics

<Processes: procedures>+≡
  subroutine process_instance_recover_hard_kinematics (instance, i_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_term
    integer :: i
    if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
      call instance%term(i_term)%recover_hard_kinematics (instance%component)
      do i = 1, size (instance%term)
        if (i /= i_term) then
          if (instance%term(i)%active) then
            call instance%term(i)% &
              compute_eff_kinematics (instance%component)
          end if
        end if
      end do
      instance%evaluation_status = STAT_EFF_KINEMATICS
    else
      ! error?
    end if
  end subroutine process_instance_recover_hard_kinematics

```

```

end if
end subroutine process_instance_recover_hard_kinematics

```

Fourth step of process evaluation: check cuts for all terms. Where successful, compute any scales and weights. Otherwise, deactivate the term. If any of the terms has passed, set the state to STAT\_PASSED\_CUTS.

```

(Processes: process instance: TBP)+≡
  procedure :: evaluate_expressions => &
    process_instance_evaluate_expressions

(Processes: procedures)+≡
  subroutine process_instance_evaluate_expressions (instance)
    class(process_instance_t), intent(inout) :: instance
    integer :: i
    if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
      do i = 1, size (instance%term)
        if (instance%term(i)%active) then
          call instance%term(i)%evaluate_expressions ()
        end if
      end do
      if (any (instance%term%passed)) then
        instance%evaluation_status = STAT_PASSED_CUTS
      else
        instance%evaluation_status = STAT_FAILED_CUTS
      end if
    else
      ! failed kinematics
    end if
  end subroutine process_instance_evaluate_expressions

```

Fifth step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

```

(Processes: process instance: TBP)+≡
  procedure :: evaluate_trace => process_instance_evaluate_trace

(Processes: procedures)+≡
  subroutine process_instance_evaluate_trace (instance)
    class(process_instance_t), intent(inout) :: instance
    integer :: i
    if (instance%evaluation_status >= STAT_PASSED_CUTS) then
      do i = 1, size (instance%term)
        if (instance%term(i)%passed) then
          call instance%term(i)%evaluate_interaction (instance%component)
          call instance%term(i)%evaluate_trace ()
        end if
      end do
      instance%sqme = 0
      do i = 1, size (instance%component)
        if (instance%component(i)%active) then
          call instance%component(i)%evaluate_sqme (instance%term)
          instance%sqme = instance%sqme + instance%component(i)%sqme
        end if
      end do
    end if
  end subroutine process_instance_evaluate_trace

```

```

        end do
        instance%evaluation_status = STAT_EVALUATED_TRACE
    else
        ! failed kinematics, failed cuts
        instance%sqme = 0
    end if
end subroutine process_instance_evaluate_trace

```

Final step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

If **weight** is provided, we already know the kinematical event weight (the MCI weight which depends on the kinematics sampling algorithm, but not on the matrix element), so we do not need to take it from the MCI record.

```

<Processes: process instance: TBP>+≡
    procedure :: evaluate_event_data => process_instance_evaluate_event_data

<Processes: procedures>+≡
    subroutine process_instance_evaluate_event_data (instance, weight)
    class(process_instance_t), intent(inout) :: instance
    real(default), intent(in), optional :: weight
    integer :: i
    if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
        do i = 1, size (instance%term)
            if (instance%term(i)%passed) then
                call instance%term(i)%evaluate_event_data ()
            end if
        end do
        if (present (weight)) then
            instance%weight = weight
        else
            instance%weight = &
                instance%mci_work(instance%i_mci)%mci%get_event_weight ()
        end if
        instance%evaluation_status = STAT_EVENT_COMPLETE
    else
        ! failed kinematics etc.
        instance%weight = 0
    end if
end subroutine process_instance_evaluate_event_data

```

For unweighted event generation, we should reset the reported event weight to unity (signed) or zero. The latter case is appropriate for an event which failed for whatever reason.

```

<Processes: process instance: TBP>+≡
    procedure :: normalize_weight => process_instance_normalize_weight

<Processes: procedures>+≡
    subroutine process_instance_normalize_weight (instance)
    class(process_instance_t), intent(inout) :: instance
    if (instance%weight /= 0) then
        instance%weight = sign (1._default, instance%weight)
    end if

```

```
end subroutine process_instance_normalize_weight
```

This is a convenience routine that performs the computations of the steps 1 to 5 in a single step. The arguments are the input for `set_mcp`. After this, the evaluation status should be either `STAT_FAILED_KINEMATICS`, `STAT_FAILED_CUTS` or `STAT_EVALUATED_TRACE`.

Before calling this, we should call `choose_mci`.

```
(Processes: process instance: TBP)+≡
```

```
procedure :: evaluate_sqme => process_instance_evaluate_sqme
```

```
(Processes: procedures)+≡
```

```
subroutine process_instance_evaluate_sqme (instance, channel, x)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: channel
  real(default), dimension(:), intent(in) :: x
  call instance%reset ()
  call instance%set_mcp (x)
  call instance%select_channel (channel)
  call instance%compute_seed_kinematics ()
  call instance%compute_hard_kinematics ()
  call instance%compute_eff_kinematics ()
  call instance%evaluate_expressions ()
  call instance%evaluate_trace ()
end subroutine process_instance_evaluate_sqme
```

This is the inverse. Assuming that the final trace evaluator contains a valid momentum configuration, recover kinematics and recalculate the matrix elements and their trace.

To be precise, we first recover kinematics for the given term and associated component, then recalculate from that all other terms and active components. The `channel` is not really required to obtain the matrix element, but it allows us to reconstruct the exact MC parameter set that corresponds to the given phase space point.

Before calling this, we should call `choose_mci`.

```
(Processes: process instance: TBP)+≡
```

```
procedure :: recover => process_instance_recover
```

```
(Processes: procedures)+≡
```

```
subroutine process_instance_recover (instance, channel, i_term, update_sqme)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: channel
  integer, intent(in) :: i_term
  logical, intent(in) :: update_sqme
  call instance%activate ()
  instance%evaluation_status = STAT_EFF_KINEMATICS
  call instance%recover_hard_kinematics (i_term)
  call instance%recover_seed_kinematics (i_term)
  call instance%select_channel (channel)
  call instance%recover_mcp (i_term)
  call instance%compute_seed_kinematics (i_term)
  call instance%compute_hard_kinematics (i_term)
  call instance%compute_eff_kinematics (i_term)
  call instance%evaluate_expressions ()
```

```

        if (update_sqme) call instance%evaluate_trace ()
    end subroutine process_instance_recover

```

The `evaluate` method is required by the `sampler_t` base type of which the process instance is an extension.

The requirement is that after the process instance is evaluated, the integrand, the selected channel, the  $x$  array, and the  $f$  Jacobian array are exposed by the `sampler_t` object.

```

<Processes: process instance: TBP>+≡
    procedure :: evaluate => process_instance_evaluate

<Processes: procedures>+≡
    subroutine process_instance_evaluate (sampler, c, x_in, val, x, f)
        class(process_instance_t), intent(inout) :: sampler
        integer, intent(in) :: c
        real(default), dimension(:), intent(in) :: x_in
        real(default), intent(out) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        call sampler%evaluate_sqme (c, x_in)
        call sampler%fetch (val, x, f)
    end subroutine process_instance_evaluate

```

The `rebuild` method should rebuild the kinematics section out of the `x_in` parameter set. The integrand value `val` should not be computed, but is provided as input.

```

<Processes: process instance: TBP>+≡
    procedure :: rebuild => process_instance_rebuild

<Processes: procedures>+≡
    subroutine process_instance_rebuild (sampler, c, x_in, val, x, f)
        class(process_instance_t), intent(inout) :: sampler
        integer, intent(in) :: c
        real(default), dimension(:), intent(in) :: x_in
        real(default), intent(in) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        call msg_bug ("process_instance_rebuild not implemented yet")
    end subroutine process_instance_rebuild

```

This is another method required by the `sampler_t` base type: fetch the data that are relevant for the MCI record.

```

<Processes: process instance: TBP>+≡
    procedure :: fetch => process_instance_fetch

<Processes: procedures>+≡
    subroutine process_instance_fetch (sampler, val, x, f)
        class(process_instance_t), intent(in) :: sampler
        real(default), intent(out) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        integer :: i, cc
        integer :: n_channel

```



```

val = 0
FIND_COMPONENT: do i = 1, size (sampler%component)
  associate (component => sampler%component(i))
    if (component%active) then
      associate (k => component%k_seed)
        n_channel = k%n_channel
        do cc = 1, n_channel
          call k%get_mcpair (cc, x(:,cc))
        end do
        f = k%f
        val = sampler%sqme * k%phs_factor
      end associate
    end FIND_COMPONENT
  end if
end associate
end do FIND_COMPONENT
end subroutine process_instance_fetch

```

### 15.5.12 Accessing the process instance

Once the seed kinematics is complete, we can retrieve the MC input parameters for all channels, not just the seed channel.

Note: We choose the first active component. This makes sense only if the seed kinematics is identical for all active components.

*(Processes: process instance: TBP)+≡*

```

procedure :: get_mcpair => process_instance_get_mcpair

```

*(Processes: procedures)+≡*

```

subroutine process_instance_get_mcpair (instance, channel, x)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: channel
  real(default), dimension(:), intent(out) :: x
  integer :: i
  if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
    do i = 1, size (instance%component)
      if (instance%component(i)%active) then
        call instance%component(i)%k_seed%get_mcpair (channel, x)
        return
      end if
    end do
    x = 0
    ! error?
  else
    x = 0
    ! error?
  end if
end subroutine process_instance_get_mcpair

```

Return true if the event is complete. In particular, the event must be kinematically valid, passed all cuts, and the event data have been computed.

*(Processes: process instance: TBP)+≡*

```

procedure :: is_complete_event => process_instance_is_complete_event

```

```

<Processes: procedures>+=
function process_instance_is_complete_event (instance) result (flag)
  class(process_instance_t), intent(in) :: instance
  logical :: flag
  flag = instance%evaluation_status >= STAT_EVENT_COMPLETE
end function process_instance_is_complete_event

```

Select a term for the process instance which is to provide the event record.

Note: this should be done using random numbers and applying probabilities for the various terms and components. The current implementation simply selects the first term for the first active component.

```

<Processes: process instance: TBP>+=
  procedure :: select_i_term => process_instance_select_i_term

<Processes: procedures>+=
  subroutine process_instance_select_i_term (instance, i_term)
    class(process_instance_t), intent(in) :: instance
    integer, intent(out) :: i_term
    integer :: i_mci, i_component
    i_mci = instance%i_mci
    i_component = instance%process%mci_entry(i_mci)%i_component(1)
    i_term = instance%process%component(i_component)%i_term(1)
  end subroutine process_instance_select_i_term

```

Return pointers to the matrix and flows interactions, given a term index.

```

<Processes: process instance: TBP>+=
  procedure :: get_trace_int_ptr => process_instance_get_trace_int_ptr
  procedure :: get_matrix_int_ptr => process_instance_get_matrix_int_ptr
  procedure :: get_flows_int_ptr => process_instance_get_flows_int_ptr

<Processes: procedures>+=
  function process_instance_get_trace_int_ptr (instance, i_term) result (ptr)
    class(process_instance_t), intent(in), target :: instance
    integer, intent(in) :: i_term
    type(interaction_t), pointer :: ptr
    ptr => instance%term(i_term)%connected%get_trace_int_ptr ()
  end function process_instance_get_trace_int_ptr

  function process_instance_get_matrix_int_ptr (instance, i_term) result (ptr)
    class(process_instance_t), intent(in), target :: instance
    integer, intent(in) :: i_term
    type(interaction_t), pointer :: ptr
    ptr => instance%term(i_term)%connected%get_matrix_int_ptr ()
  end function process_instance_get_matrix_int_ptr

  function process_instance_get_flows_int_ptr (instance, i_term) result (ptr)
    class(process_instance_t), intent(in), target :: instance
    integer, intent(in) :: i_term
    type(interaction_t), pointer :: ptr
    ptr => instance%term(i_term)%connected%get_flows_int_ptr ()
  end function process_instance_get_flows_int_ptr

```

Return the indices of the beam particles and incoming partons within the currently active state matrix, respectively.

```

(Processes: process instance: TBP)+≡
  procedure :: get_beam_index => process_instance_get_beam_index
  procedure :: get_in_index => process_instance_get_in_index

(Processes: procedures)+≡
  subroutine process_instance_get_beam_index (instance, i_term, i_beam)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    integer, dimension(:), intent(out) :: i_beam
    call instance%term(i_term)%connected%get_beam_index (i_beam)
  end subroutine process_instance_get_beam_index

  subroutine process_instance_get_in_index (instance, i_term, i_in)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    integer, dimension(:), intent(out) :: i_in
    call instance%term(i_term)%connected%get_in_index (i_in)
  end subroutine process_instance_get_in_index

```

Return squared matrix element and event weight.

```

(Processes: process instance: TBP)+≡
  procedure :: get_sqme => process_instance_get_sqme
  procedure :: get_weight => process_instance_get_weight

(Processes: procedures)+≡
  function process_instance_get_sqme (instance) result (sqme)
    class(process_instance_t), intent(in) :: instance
    real(default) :: sqme
    if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
      sqme = instance%sqme
    else
      sqme = 0    ! error?
    end if
  end function process_instance_get_sqme

  function process_instance_get_weight (instance) result (weight)
    class(process_instance_t), intent(in) :: instance
    real(default) :: weight
    if (instance%evaluation_status >= STAT_EVENT_COMPLETE) then
      weight = instance%weight
    else
      weight = 0    ! error?
    end if
  end function process_instance_get_weight

```

Return the currently selected MCI channel.

```

(Processes: process instance: TBP)+≡
  procedure :: get_channel => process_instance_get_channel

(Processes: procedures)+≡
  function process_instance_get_channel (instance) result (channel)
    class(process_instance_t), intent(in) :: instance

```

```

integer :: channel
channel = instance%selected_channel
end function process_instance_get_channel

```

Return factorization scale and strong coupling. We have to select a term instance.

```

<Processes: process instance: TBP>+≡
  procedure :: get_fac_scale => process_instance_get_fac_scale
  procedure :: get_alpha_s => process_instance_get_alpha_s

<Processes: procedures>+≡
  function process_instance_get_fac_scale (instance, i_term) result (fac_scale)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    real(default) :: fac_scale
    fac_scale = instance%term(i_term)%get_fac_scale ()
  end function process_instance_get_fac_scale

  function process_instance_get_alpha_s (instance, i_term) result (alpha_s)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    real(default) :: alpha_s
    alpha_s = instance%term(i_term)%get_alpha_s (instance%component)
  end function process_instance_get_alpha_s

```

### 15.5.13 Particle sets

Here we provide two procedures that convert the process instance from/to a particle set. The conversion applies to the trace evaluator which has no quantum-number information, thus it involves only the momenta and the parent-child relations. We keep virtual particles.

Nevertheless, it is possible to reconstruct the complete structure from a particle set. The reconstruction implies a re-evaluation of the structure function and matrix-element codes.

The `i_term` index is needed for both input and output, to select among different active trace evaluators.

In both cases, the `instance` object must be properly initialized.

```

<Processes: process instance: TBP>+≡
  procedure :: get_trace => process_instance_get_trace
  procedure :: set_trace => process_instance_set_trace

<Processes: procedures>+≡
  subroutine process_instance_get_trace (instance, pset, i_term)
    class(process_instance_t), intent(in), target :: instance
    type(particle_set_t), intent(out) :: pset
    integer, intent(in) :: i_term
    type(interaction_t), pointer :: int
    logical :: ok
    int => instance%get_trace_int_ptr (i_term)
    call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
      [0._default, 0._default], .false., .true.)
  end subroutine process_instance_get_trace

```

```

subroutine process_instance_set_trace (instance, pset, i_term)
  class(process_instance_t), intent(inout), target :: instance
  type(particle_set_t), intent(in) :: pset
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: int
  integer :: n_in
  int => instance%get_trace_int_ptr (i_term)
  n_in = instance%process%get_n_in ()
  call particle_set_fill_interaction (pset, int, n_in)
end subroutine process_instance_set_trace

```

### 15.5.14 Auxiliary stuff

Write a separator line.

```

<Processes: procedures>+≡
  subroutine write_separator (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat (" ", 72)
  end subroutine write_separator

  subroutine write_separator_double (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("=", 72)
  end subroutine write_separator_double

```

### 15.5.15 Unit tests

```

<Processes: public>+≡
  public :: processes_test

<Processes: tests>≡
  subroutine processes_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <Processes: execute tests>
  end subroutine processes_test

```

### Test process type

For the following tests, we define a simple implementation of the abstract `prc_core_t`, designed such as to complement the `prc_test_t` process definition type.

Note that it is not given that the actual process is defined as `prc_test_t` type. We enforce this by calling `prc_test_create_library`. The driver component in the process core will then become of type `prc_test_t`.

```

<Processes: public>+≡
  public :: test_t

```

```

<Processes: test types>≡
  type, extends (prc_core_t) :: test_t
  contains
    procedure :: write => test_write
    procedure :: needs_mcset => test_needs_mcset
    procedure :: get_n_terms => test_get_n_terms
    procedure :: is_allowed => test_is_allowed
    procedure :: compute_hard_kinematics => test_compute_hard_kinematics
    procedure :: compute_eff_kinematics => test_compute_eff_kinematics
    procedure :: recover_kinematics => test_recover_kinematics
    procedure :: compute_amplitude => test_compute_amplitude
  end type test_t

<Processes: tests>+≡
  subroutine test_write (object, unit)
    class(test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A)") "test type implementing prc_test"
  end subroutine test_write

```

This process type always needs a MC parameter set and a single term. This only state is always allowed.

```

<Processes: tests>+≡
  function test_needs_mcset (object) result (flag)
    class(test_t), intent(in) :: object
    logical :: flag
    flag = .true.
  end function test_needs_mcset

  function test_get_n_terms (object) result (n)
    class(test_t), intent(in) :: object
    integer :: n
    n = 1
  end function test_get_n_terms

  function test_is_allowed (object, i_term, f, h, c) result (flag)
    class(test_t), intent(in) :: object
    integer, intent(in) :: i_term, f, h, c
    logical :: flag
    flag = .true.
  end function test_is_allowed

```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

```

<Processes: tests>+≡
  subroutine test_compute_hard_kinematics &
    (object, p_seed, i_term, int_hard, tmp)
    class(test_t), intent(in) :: object
    type(vector4_t), dimension(:), intent(in) :: p_seed
    integer, intent(in) :: i_term

```

```

type(interaction_t), intent(inout) :: int_hard
class(workspace_t), intent(inout), allocatable :: tmp
call interaction_set_momenta (int_hard, p_seed)
end subroutine test_compute_hard_kinematics

```

This procedure is not called for `test_t`, just a placeholder.

```

<Processes: tests>+≡
subroutine test_compute_eff_kinematics &
  (object, i_term, int_hard, int_eff, tmp)
class(test_t), intent(in) :: object
integer, intent(in) :: i_term
type(interaction_t), intent(in) :: int_hard
type(interaction_t), intent(inout) :: int_eff
class(workspace_t), intent(inout), allocatable :: tmp
end subroutine test_compute_eff_kinematics

```

Transfer the incoming momenta of `p_seed` directly to the effective interaction, and vice versa for the outgoing momenta.

`int_hard` is left untouched since `int_eff` is an alias (via pointer) to it.

```

<Processes: tests>+≡
subroutine test_recover_kinematics &
  (object, p_seed, int_hard, int_eff, tmp)
class(test_t), intent(in) :: object
type(vector4_t), dimension(:), intent(inout) :: p_seed
type(interaction_t), intent(inout) :: int_hard
type(interaction_t), intent(inout) :: int_eff
class(workspace_t), intent(inout), allocatable :: tmp
integer :: n_in
n_in = interaction_get_n_in (int_eff)
call interaction_set_momenta (int_eff, p_seed(1:n_in), outgoing = .false.)
p_seed(n_in+1:) = interaction_get_momenta (int_eff, outgoing = .true.)
end subroutine test_recover_kinematics

```

Compute the amplitude. The driver ignores all quantum numbers and, in fact, returns a constant. Nevertheless, we properly transfer the momentum vectors.

```

<Processes: tests>+≡
function test_compute_amplitude &
  (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
class(test_t), intent(in) :: object
integer, intent(in) :: j
type(vector4_t), dimension(:), intent(in) :: p
integer, intent(in) :: f, h, c
real(default), intent(in) :: fac_scale, ren_scale
class(workspace_t), intent(inout), allocatable, optional :: tmp
complex(default) :: amp
real(default), dimension(:,:), allocatable :: parray
integer :: i
select type (driver => object%driver)
type is (prc_test_t)
  allocate (parray (0:3,4))
  forall (i = 1:4) parray(:,i) = vector4_get_components (p(i))
  amp = driver%get_amplitude (parray)

```

```

end select
end function test_compute_amplitude

```

## Write an empty process object

The most trivial test is to write an uninitialized process object.

```

<Processes: execute tests>≡
  call test (processes_1, "processes_1", &
    "write an empty process object", &
    u, results)

<Processes: tests>+≡
  subroutine processes_1 (u)
    integer, intent(in) :: u
    type(process_t) :: process

    write (u, "(A)")  "* Test output: processes_1"
    write (u, "(A)")  "* Purpose: display an empty process object"
    write (u, "(A)")

    call process%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_1"

  end subroutine processes_1

```

## Initialize a process object

Initialize a process and display it.

```

<Processes: execute tests>+≡
  call test (processes_2, "processes_2", &
    "initialize a simple process object", &
    u, results)

<Processes: tests>+≡
  subroutine processes_2 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template

    write (u, "(A)")  "* Test output: processes_2"
    write (u, "(A)")  "* Purpose: initialize a simple process object"
    write (u, "(A)")

```



```

write (u, "(A)")  "* Build and load a test library with one process"
write (u, "(A)")

libname = "processes2"
procname = libname
run_id = "run2"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

write (u, "(A)")  "* Initialize a process object"
write (u, "(A)")

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

call process%setup_mci ()

call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_2"

end subroutine processes_2

```

### Compute a trivial matrix element

Initialize a process, retrieve some information and compute a matrix element.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

```

<Processes: execute tests>+≡
    call test (processes_3, "processes_3", &
        "retrieve a trivial matrix element", &
        u, results)

<Processes: tests>+≡
    subroutine processes_3 (u)
        integer, intent(in) :: u
        type(process_library_t), target :: lib
        type(string_t) :: libname

```

```

type(string_t) :: procname
type(string_t) :: run_id
type(os_data_t) :: os_data
type(process_t), allocatable :: process
class(prc_core_t), allocatable :: core_template
class(mci_t), allocatable :: mci_template
class(phs_config_t), allocatable :: phs_config_template
type(process_constants_t) :: data
type(vector4_t), dimension(:), allocatable :: p

write (u, "(A)")  "* Test output: processes_3"
write (u, "(A)")  "* Purpose: create a process &
                  &and compute a matrix element"
write (u, "(A)")

write (u, "(A)")  "* Build and load a test library with one process"
write (u, "(A)")

libname = "processes3"
procname = libname
run_id = "run3"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t)
    call mci_template%set_dimensions (2, 2)
    call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Return the number of process components"
write (u, "(A)")

write (u, "(A,IO)")  "n_components = ", process%get_n_components ()

write (u, "(A)")
write (u, "(A)")  "* Return the number of flavor states"
write (u, "(A)")

data = process%get_constants (1)

write (u, "(A,IO)")  "n_flv(1) = ", data%n_flv

write (u, "(A)")
write (u, "(A)")  "* Return the first flavor state"

```

```

write (u, "(A)")

write (u, "(A,4(1x,I0))") "flv_state(1) =", data%flv_state (:,1)

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics &
    &[arbitrary, the matrix element is constant]"

allocate (p (4))

write (u, "(A)")
write (u, "(A)")  "* Retrieve the matrix element"
write (u, "(A)")

write (u, "(A,F5.3,' + ',F5.3,' I')") "me (1, p, 1, 1, 1) = ", &
    process%compute_amplitude (1, 1, p, 1, 1, 1)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_3"

end subroutine processes_3

```

## Generate a process instance

Initialize a process and process instance, choose a sampling point and fill the process instance.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

```

<Processes: execute tests>+≡
    call test (processes_4, "processes_4", &
        "create and fill a process instance (partonic event)", &
        u, results)

<Processes: tests>+≡
subroutine processes_4 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template

```

```

class(phs_config_t), allocatable :: phs_config_template
real(default) :: sqrts
type(process_instance_t), allocatable, target :: process_instance
type(particle_set_t) :: pset

write (u, "(A)")  "* Test output: processes_4"
write (u, "(A)")  "* Purpose: create a process &
                  &and fill a process instance"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "processes4"
procname = libname
run_id = "run4"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
      (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams (sqrts)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"

```

```

write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpair ([0._default, 0._default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%evaluate_trace ()
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1)

call process_instance%activate ()
process_instance%evaluation_status = STAT_EFF_KINEMATICS
call process_instance%recover_hard_kinematics (i_term = 1)
call process_instance%recover_seed_kinematics (i_term = 1)
call process_instance%select_channel (1)
call process_instance%recover_mcpair (i_term = 1)

call process_instance%compute_seed_kinematics (skip_term = 1)
call process_instance%compute_hard_kinematics (skip_term = 1)

```

```

call process_instance%compute_eff_kinematics (skip_term = 1)

call process_instance%evaluate_expressions ()
call process_instance%evaluate_trace ()
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (pset)
call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_4"

end subroutine processes_4

```

## Handle partonic cuts

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

```

<Processes: execute tests>+≡
  call test (processes_5, "processes_5", &
    "handle cuts (partonic event)", &
    u, results)

<Processes: tests>+≡
  subroutine processes_5 (u)
    integer, intent(in) :: u
    type(string_t) :: cut_expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: parse_tree
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance

```

```

write (u, "(A)")  "* Test output: processes_5"
write (u, "(A)")  "* Purpose: create a process &
    &and fill a process instance"
write (u, "(A)")

write (u, "(A)")  "* Prepare a cut expression"
write (u, "(A)")

call syntax_pexpr_init ()
cut_expr_text = "all Pt > 100 [s]"
call ifile_append (ifile, cut_expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)

write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "processes4"
procname = libname
run_id = "run4"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, lib, os_data)
call process%set_var_list (model_get_var_list_ptr (process%config%model))

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization and set cuts"
write (u, "(A)")

call process%setup_terms ()
call process%set_cuts (parse_tree_get_root_ptr (parse_tree))
call process%write (u, show_var_list=.true., show_expressions=.true.)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

```

```

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpair ([0._default, 0._default])

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics and subevt, check cuts (should fail)"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set (should succeed)"
write (u, "(A)")

call process_instance%reset ()
call process_instance%set_mcpair ([0.5_default, 0.125_default])
call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%evaluate_trace ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
&(failure)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.0_default, 0.2_default])

call process_instance%write_header (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
&(success)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.1_default, 0.2_default])

call process_instance%write_header (u)

```



```

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

call parse_tree_final (parse_tree)
call stream_final (stream)
call ifile_final (ifile)
call syntax_pexpr_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_5"

end subroutine processes_5

```

## Scales and such

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

```

(Processes: execute tests) +=
  call test (processes_6, "processes_6", &
    "handle scales and weight (partonic event)", &
    u, results)

(Processes: tests) +=
  subroutine processes_6 (u)
    integer, intent(in) :: u
    type(string_t) :: expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_scale, pt_fac_scale, pt_ren_scale, pt_weight
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: processes_6"

```

```

write (u, "(A)")  "* Purpose: create a process &
                  &and fill a process instance"
write (u, "(A)")

write (u, "(A)")  "* Prepare expressions"
write (u, "(A)")

call syntax_pexpr_init ()

expr_text = "sqrts - 100 GeV"
write (u, "(A,A)") "scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_scale, stream, .true.)
call stream_final (stream)

expr_text = "sqrts_hat"
write (u, "(A,A)") "fac_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
call stream_final (stream)

expr_text = "eval sqrt (M2) [collect [s]]"
write (u, "(A,A)") "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)

expr_text = "n_tot * n_in * n_out * (eval Phi / pi [s])"
write (u, "(A,A)") "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "processes4"
procname = libname
run_id = "run4"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

```

```

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, lib, os_data)
call process%set_var_list (model_get_var_list_ptr (process%config%model))

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization and set cuts"
write (u, "(A)")

call process%setup_terms ()
call process%set_scale (parse_tree_get_root_ptr (pt_scale))
call process%set_fac_scale (parse_tree_get_root_ptr (pt_fac_scale))
call process%set_ren_scale (parse_tree_get_root_ptr (pt_ren_scale))
call process%set_weight (parse_tree_get_root_ptr (pt_weight))
call process%write (u, show_expressions=.true.)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance and evaluate"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, [0.5_default, 0.125_default])

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

call parse_tree_final (pt_scale)
call parse_tree_final (pt_fac_scale)
call parse_tree_final (pt_ren_scale)

```

```

call parse_tree_final (pt_weight)
call syntax_pexpr_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_6"

end subroutine processes_6

```

## Structure function configuration

Configure structure functions (multi-channel) in a process object.

```

(Processes: execute tests) +=
  call test (processes_7, "processes_7", &
    "process configuration with structure functions", &
    u, results)

(Processes: tests) +=
  subroutine processes_7 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_channel_t), dimension(2) :: sf_channel

    write (u, "(A)")  "* Test output: processes_7"
    write (u, "(A)")  "* Purpose: initialize a process with &
      &structure functions"
    write (u, "(A)")

    write (u, "(A)")  "* Build and initialize a process object"
    write (u, "(A)")

    libname = "processes7"
    procname = libname
    run_id = "run7"
    call os_data_init (os_data)
    call prc_test_create_library (libname, lib)
    call syntax_model_file_init ()

    allocate (process)
    call process%init (procname, run_id, lib, os_data)

```

```

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)
call process%configure_phs ()

model => model_list_get_model_ptr (var_str ("Test"))
call flavor_init (flv, 25, model)
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

process%beam_config%n_channel = 3

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call sf_channel(1)%init (2)
call sf_channel(1)%activate_mapping ([1,2])
call process%set_sf_channel (2, sf_channel(1))

call sf_channel(2)%init (2)
call sf_channel(2)%set_s_mapping ([1,2])
call process%set_sf_channel (3, sf_channel(2))

call process%setup_mci ()

call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_7"

end subroutine processes_7

```

## Evaluating a process with structure function

Configure structure functions (single-channel) in a process object, create an instance, compute kinematics and evaluate.

Note the order of operations when setting up structure functions and phase space. The beams are first, they determine the `sqrts` value. We can also set up the chain of structure functions. We then configure the phase space. From this, we can obtain information about special configurations (resonances, etc.), which we need for allocating the possible structure-function channels (parameterizations and mappings). Finally, we match phase-space channels onto structure-function channels.

In the current example, this matching is trivial; we only have one structure-function channel.

```
<Processes: execute tests>+≡
    call test (processes_8, "processes_8", &
               "process evaluation with structure functions", &
               u, results)

<Processes: tests>+≡
    subroutine processes_8 (u)
        integer, intent(in) :: u
        type(process_library_t), target :: lib
        type(string_t) :: libname
        type(string_t) :: procname
        type(string_t) :: run_id
        type(os_data_t) :: os_data
        type(process_t), allocatable, target :: process
        class(prc_core_t), allocatable :: core_template
        class(mci_t), allocatable :: mci_template
        class(phs_config_t), allocatable :: phs_config_template
        real(default) :: sqrts
        type(process_instance_t), allocatable, target :: process_instance
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable, target :: data
        type(sf_config_t), dimension(:), allocatable :: sf_config
        type(sf_channel_t) :: sf_channel
        type(particle_set_t) :: pset

        write (u, "(A)")  "* Test output: processes_8"
        write (u, "(A)")  "*   Purpose: evaluate a process with &
                           &structure functions"
        write (u, "(A)")

        write (u, "(A)")  "* Build and initialize a process object"
        write (u, "(A)")

        libname = "processes8"
        procname = libname
        run_id = "run8"
        call os_data_init (os_data)
        call prc_test_create_library (libname, lib)
        call syntax_model_file_init ()
```

```

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)

model => model_list_get_model_ptr (var_str ("Test"))
call flavor_init (flv, 25, model)
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (1, sf_channel)

call process%match_channels ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_mci ()
call process%setup_terms ()

call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Set up kinematics and evaluate"

```

```

write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, &
    [0.8_default, 0.8_default, 0.1_default, 0.2_default])
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1)
call process_instance%recover &
    (channel = 1, i_term = 1, update_sqme = .true.)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (pset)

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_8"

end subroutine processes_8

```



## Multi-channel phase space and structure function

This is an extension of the previous example. This time, we have two distinct structure-function channels which are matched to the two distinct phase-space channels.

```

<Processes: execute tests>+≡
    call test (processes_9, "processes_9", &
               "multichannel kinematics and structure functions", &
               u, results)

<Processes: tests>+≡
    subroutine processes_9 (u)
        integer, intent(in) :: u
        type(process_library_t), target :: lib
        type(string_t) :: libname
        type(string_t) :: procname
        type(string_t) :: run_id
        type(os_data_t) :: os_data
        type(process_t), allocatable, target :: process
        class(prc_core_t), allocatable :: core_template
        class(mci_t), allocatable :: mci_template
        class(phs_config_t), allocatable :: phs_config_template
        real(default) :: sqrts
        type(process_instance_t), allocatable, target :: process_instance
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable, target :: data
        type(sf_config_t), dimension(:), allocatable :: sf_config
        type(sf_channel_t) :: sf_channel
        real(default), dimension(4) :: x_saved
        type(particle_set_t) :: pset

        write (u, "(A)")  "* Test output: processes_9"
        write (u, "(A)")  "*   Purpose: evaluate a process with &
                           &structure functions"
        write (u, "(A)")  "*                           in a multi-channel configuration"
        write (u, "(A)")

        write (u, "(A)")  "* Build and initialize a process object"
        write (u, "(A)")

        libname = "processes9"
        procname = libname
        run_id = "run9"
        call os_data_init (os_data)
        call prc_test_create_library (libname, lib)
        call syntax_model_file_init ()

        call reset_interaction_counter ()

        allocate (process)
        call process%init (procname, run_id, lib, os_data)

        allocate (test_t :: core_template)
        allocate (phs_test_config_t :: phs_config_template)

```

```

call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams (sqrts)

model => model_list_get_model_ptr (var_str ("Test"))
call flavor_init (flv, 25, model)
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
    call data%init (model, flv)
end select

process%beam_config%n_channel = 2

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call sf_channel%init (2)
call process%set_sf_channel (1, sf_channel)

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (2, sf_channel)

call process%match_channels ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_mci ()
call process%setup_terms ()

call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Set up kinematics in channel 1 and evaluate"
write (u, "(A)")

```

```

call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, &
    [0.8_default, 0.8_default, 0.1_default, 0.2_default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract MC input parameters"
write (u, "(A)")

write (u, "(A)")  "Channel 1:"
call process_instance%get_mcpair (1, x_saved)
write (u, "(2x,9(1x,F7.5))") x_saved

write (u, "(A)")  "Channel 2:"
call process_instance%get_mcpair (2, x_saved)
write (u, "(2x,9(1x,F7.5))") x_saved

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics in channel 2 and evaluate"
write (u, "(A)")

call process_instance%evaluate_sqme (2, x_saved)
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance for channel 2"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1)
call process_instance%recover &
    (channel = 2, i_term = 1, update_sqme = .true.)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (pset)

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

```

```

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "** Test output end: processes_9"

end subroutine processes_9

```

## Event generation

Activate the MC integrator for the process object and use it to generate a single event. Note that the test integrator does not require integration in preparation for generating events.

```

(Processes: execute tests) +=
  call test (processes_10, "processes_10", &
    "event generation", &
    u, results)

(Processes: tests) +=
  subroutine processes_10 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "** Test output: processes_10"
    write (u, "(A)")  "** Purpose: generate events for a process without &
      &structure functions"
    write (u, "(A)")  "**           in a multi-channel configuration"
    write (u, "(A)")

    write (u, "(A)")  "** Build and initialize a process object"
    write (u, "(A)")

    libname = "processes10"
    procname = libname
    run_id = "run10"
    call os_data_init (os_data)
    call prc_test_create_library (libname, lib)
    call syntax_model_file_init ()

    call reset_interaction_counter ()

    allocate (process)
    call process%init (procname, run_id, lib, os_data)

```

```

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t); call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)
call process%configure_phs ()

call process%setup_mci ()
allocate (rng_test_t :: rng)
call process%import_rng (1, rng)

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

select type (mci => process%mci_entry(1)%mci)
type is (mci_test_t)
    ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
    call mci%rng%init (3)
    ! Include the constant PHS factor in the stored maximum of the integrand
    call mci%set_max_factor (conv * twopi4 &
        / (2 * sqrt (lambda (sqrt_s **2, 125._default**2, 125._default**2))))
end select

call process%generate_weighted_event (process_instance, 1)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call process%generate_unweighted_event (process_instance, 1)

```

```

select type (mci => process%mci_entry(1)%mci)
type is (mci_test_t)
    write (u, "(A,I0)") " Success in try ", mci%tries
    write (u, "(A)")
end select

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_10"

end subroutine processes_10

```

## Integration

Activate the MC integrator for the process object and use it to integrate over phase space.

```

<Processes: execute tests>+≡
    call test (processes_11, "processes_11", &
        "integration", &
        u, results)

<Processes: tests>+≡
subroutine processes_11 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(process_t), allocatable, target :: process
    class(prc_core_t), allocatable :: core_template
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: processes_11"
    write (u, "(A)")  "* Purpose: integrate a process without &
        &structure functions"
    write (u, "(A)")  "*           in a multi-channel configuration"
    write (u, "(A)")

```

```

write (u, "(A)")  "* Build and initialize a process object"
write (u, "(A)")

libname = "processes11"
procname = libname
run_id = "run11"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t)
    call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrt_s = 1000
call process%setup_beams (sqrt_s)
call process%configure_phs ()

call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Integrate with default test parameters"
write (u, "(A)")

call process%integrate (process_instance, 1, n_it=1, n_calls=10000)
call process%final_integration (1)

```

```

call process%write (u)

write (u, "(A)")
write (u, "(A,ES13.7)") " Integral divided by phs factor = ", &
    process%get_integral (1) &
    / process_instance%component(1)%k_seed%phs_factor

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_11"

end subroutine processes_11

```

## Complete events

For the purpose of simplifying further tests, we implement a convenience routine that initializes a process and prepares a single event. This is a wrapup of the test `processes_10`.

```

<Processes: public>+≡
    public :: prepare_test_process

<Processes: procedures>+≡
    subroutine prepare_test_process (process, process_instance)
        type(process_t), intent(out), target :: process
        type(process_instance_t), intent(out), target :: process_instance
        type(process_library_t), target :: lib
        type(string_t) :: libname
        type(string_t) :: procname
        type(string_t) :: run_id
        type(os_data_t) :: os_data
        class(prc_core_t), allocatable :: core_template
        class(mci_t), allocatable :: mci_template
        class(phs_config_t), allocatable :: phs_config_template
        real(default) :: sqrts
        class(rng_t), allocatable :: rng
        libname = "processes_test"
        procname = libname
        run_id = "run_test"
        call os_data_init (os_data)
        call prc_test_create_library (libname, lib)
        call syntax_model_file_init ()
        call reset_interaction_counter ()
        call process%init (procname, run_id, lib, os_data)
    end subroutine

```



```

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t); call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
    (1, core_template, mci_template, phs_config_template)
sqrts = 1000
call process%setup_beams (sqrts)
call process%configure_phs ()
call process%setup_mci ()
allocate (rng_test_t :: rng)
call process%import_rng (1, rng)
call process%setup_terms ()
call process_instance%init (process)
select type (mci => process%mci_entry(1)%mci)
type is (mci_test_t)
    ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
    call mci%rng%init (3)
    ! Include the constant PHS factor in the stored maximum of the integrand
    call mci%set_max_factor (conv * twopi4 &
        / (2 * sqrt (lambda (sqrts **2, 125._default**2, 125._default**2))))
end select
end subroutine prepare_test_process

```

Here we do the cleanup of the process and process instance emitted by the previous routine.

```

<Processes: public>+≡
    public :: cleanup_test_process

<Processes: procedures>+≡
    subroutine cleanup_test_process (process, process_instance)
        type(process_t), intent(inout) :: process
        type(process_instance_t), intent(inout) :: process_instance
        call model_list_final ()
        call process_instance%final ()
        call process%final ()
    end subroutine cleanup_test_process

```

This is the actual test. Prepare the test process and event, fill all evaluators, and display the results. Use a particle set as temporary storage, read kinematics and recalculate the event.

```

<Processes: execute tests>+≡
    call test (processes_12, "processes_12", &
        "event post-processing", &
        u, results)

<Processes: tests>+≡
    subroutine processes_12 (u)
        integer, intent(in) :: u
        type(process_t), allocatable, target :: process
        type(process_instance_t), allocatable, target :: process_instance
        type(particle_set_t) :: pset

```

```

write (u, "(A)")  "* Test output: processes_12"
write (u, "(A)")  "* Purpose: generate a complete partonic event"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize process and process instance &
                    &and generate event"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance)
call process_instance%setup_event_data ()

call process%init_simulation (process_instance, 1)
call process%generate_weighted_event (process_instance, 1)
call process_instance%evaluate_event_data ()

call process_instance%write (u)

call process_instance%get_trace (pset, 1)

call process%final_simulation (process_instance, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Recover kinematics and recalculate"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1)
call process_instance%recover &
    (channel = 1, i_term = 1, update_sqme = .true.)

call process%recover_event (process_instance, 1)
call process_instance%evaluate_event_data ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_12"

end subroutine processes_12

```

### Colored interaction

This test specifically checks the transformation of process data (flavor, helicity, and color) into an interaction in a process term.

We use the `test_t` process core (which has no nontrivial particles), but call only the `is_allowed` method, which always returns true.

```

<Processes: execute tests>+≡
  call test (processes_13, "processes_13", &
    "colored interaction", &
    u, results)

<Processes: tests>+≡
  subroutine processes_13 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(process_term_t) :: term
    class(prc_core_t), allocatable :: core

    write (u, "(A)")  "* Test output: processes_12"
    write (u, "(A)")  "* Purpose: initialized a colored interaction"
    write (u, "(A)")

    write (u, "(A)")  "* Set up a process constants block"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list_read_model (var_str ("QCD"), var_str ("QCD.mdl"), &
      os_data, model)
    allocate (test_t :: core)

    associate (data => term%data)
      data%n_in = 2
      data%n_out = 3
      data%n_flv = 2
      data%n_hel = 2
      data%n_col = 2
      data%n_cin = 2

      allocate (data%flv_state (5, 2))
      data%flv_state (:,1) = [ 1, 21, 1, 21, 21]
      data%flv_state (:,2) = [ 2, 21, 2, 21, 21]

      allocate (data%hel_state (5, 2))
      data%hel_state (:,1) = [1, 1, 1, 1, 0]
      data%hel_state (:,2) = [1,-1, 1,-1, 0]
    end associate
  end subroutine

```

```

allocate (data%col_state (2, 5, 2))
data%col_state (:,:,1) = &
    reshape ([[1, 0], [2,-1], [3, 0], [2,-3], [0,0]], [2,5])
data%col_state (:,:,2) = &
    reshape ([[1, 0], [2,-3], [3, 0], [2,-1], [0,0]], [2,5])

allocate (data%ghost_flag (5, 2))
data%ghost_flag(1:4,:) = .false.
data%ghost_flag(5,:) = .true.

end associate

write (u, "(A)")  "* Set up the interaction"
write (u, "(A)")

call reset_interaction_counter ()
call term%setup_interaction (core, model)
call interaction_write (term%int, u)

call model_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_13"
end subroutine processes_13

```

## MD5 sums

Configure a process with structure functions (multi-channel) and compute MD5 sums

```

<Processes: execute tests>+≡
    call test (processes_14, "processes_14", &
        "process configuration and MD5 sum", &
        u, results)

<Processes: tests>+≡
    subroutine processes_14 (u)
        integer, intent(in) :: u
        type(process_library_t), target :: lib
        type(string_t) :: libname
        type(string_t) :: procname
        type(string_t) :: run_id
        type(os_data_t) :: os_data
        type(process_t), allocatable, target :: process
        class(prc_core_t), allocatable :: core_template
        class(mci_t), allocatable :: mci_template
        class(phs_config_t), allocatable :: phs_config_template
        real(default) :: sqrts
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        class(sf_data_t), allocatable, target :: data
        type(sf_config_t), dimension(:), allocatable :: sf_config
        type(sf_channel_t), dimension(2) :: sf_channel

```

```

write (u, "(A)")  "* Test output: processes_14"
write (u, "(A)")  "* Purpose: initialize a process with &
                  &structure functions"
write (u, "(A)")  "*                  and compute MD5 sum"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize a process object"
write (u, "(A)")

libname = "processes7"
procname = libname
run_id = "run7"
call os_data_init (os_data)
call prc_test_create_library (libname, lib)
call lib%compute_md5sum ()

call syntax_model_file_init ()

allocate (process)
call process%init (procname, run_id, lib, os_data)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
      (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams (sqrts)
call process%configure_phs ()

model => model_list_get_model_ptr (var_str ("Test"))
call flavor_init (flv, 25, model)
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
      call data%init (model, flv)
end select

process%beam_config%n_channel = 3

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call sf_channel(1)%init (2)
call sf_channel(1)%activate_mapping ([1,2])
call process%set_sf_channel (2, sf_channel(1))

```

```

call sf_channel(2)%init (2)
call sf_channel(2)%set_s_mapping ([1,2])
call process%set_sf_channel (3, sf_channel(2))

call process%setup_mci ()

call process%compute_md5sum ()

call process%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process%final ()
deallocate (process)

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: processes_14"

end subroutine processes_14

```

## 15.6 Process Stacks

For storing and handling multiple processes, we define process stacks. These are ordinary stacks where new process entries are pushed onto the top. We allow for multiple entries with identical process ID, but distinct run ID.

The implementation is essentially identical to the `prclib_stacks` module above. Unfortunately, Fortran supports no generic programming, so we do not make use of this fact.

When searching for a specific process ID, we will get (a pointer to) the top-most process entry with that ID on the stack, which was entered last. Usually, this is the best version of the process (in terms of integral, etc.) Thus the stack terminology makes sense.

```

<process_stacks.f90>≡
  <File header>

  module process_stacks

    <Use strings>
    <Use file utils>
    use unit_tests
    use os_interface
    use models

    use process_libraries
    use prc_test
    use processes

    <Standard module head>

```

```

    <Process stacks: public>

    <Process stacks: types>

    contains

    <Process stacks: procedures>

    <Process stacks: tests>

    end module process_stacks

```

### 15.6.1 The process entry type

A process entry is a process object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, processes should be prepared as process entry objects.

```

    <Process stacks: public>≡
        public :: process_entry_t

    <Process stacks: types>≡
        type, extends (process_t) :: process_entry_t
            type(process_entry_t), pointer :: next => null ()
        end type process_entry_t

```

### 15.6.2 The process stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

```

    <Process stacks: public>+≡
        public :: process_stack_t

    <Process stacks: types>+≡
        type :: process_stack_t
            integer :: n = 0
            type(process_entry_t), pointer :: first => null ()
        contains
            <Process stacks: process stack: TBP>
        end type process_stack_t

```

Finalizer. Iteratively deallocate the stack entries. The resulting empty stack can be immediately recycled, if necessary.

```

    <Process stacks: process stack: TBP>≡
        procedure :: final => process_stack_final

    <Process stacks: procedures>≡
        subroutine process_stack_final (object)
            class(process_stack_t), intent(inout) :: object
            type(process_entry_t), pointer :: process
            do while (associated (object%first))
                process => object%first
            end do
        end subroutine

```

```

        object%first => process%next
        call process%final ()
        deallocate (process)
    end do
    object%n = 0
end subroutine process_stack_final

```

Output. The processes on the stack will be ordered LIFO, i.e., backwards.

```

(Process stacks: process stack: TBP)+≡
    procedure :: write => process_stack_write

(Process stacks: procedures)+≡
    subroutine process_stack_write (object, unit)
        class(process_stack_t), intent(in) :: object
        integer, intent(in), optional :: unit
        type(process_entry_t), pointer :: process
        integer :: u
        u = output_unit (unit)
        call write_separator_double (u)
        select case (object%n)
        case (0)
            write (u, "(1x,A)") "Process stack: [empty]"
            call write_separator_double (u)
        case default
            write (u, "(1x,A)") "Process stack:"
            process => object%first
            do while (associated (process))
                call process%write (u)
                process => process%next
            end do
        end select
    end subroutine process_stack_write

```

### 15.6.3 Push

We take a process pointer and push it onto the stack. The previous pointer is nullified. Subsequently, the process is ‘owned’ by the stack and will be finalized when the stack is deleted.

```

(Process stacks: process stack: TBP)+≡
    procedure :: push => process_stack_push

(Process stacks: procedures)+≡
    subroutine process_stack_push (stack, process)
        class(process_stack_t), intent(inout) :: stack
        type(process_entry_t), intent(inout), pointer :: process
        process%next => stack%first
        stack%first => process
        process => null ()
        stack%n = stack%n + 1
    end subroutine process_stack_push

```



### 15.6.4 Data Access

Return a pointer to a process with specific ID.

```
<Process stacks: process_stack: TBP>+≡
    procedure :: get_process_ptr => process_stack_get_process_ptr

<Process stacks: procedures>+≡
    function process_stack_get_process_ptr (stack, id) result (ptr)
        class(process_stack_t), intent(in) :: stack
        type(string_t), intent(in) :: id
        type(process_t), pointer :: ptr
        type(process_entry_t), pointer :: entry
        ptr => null ()
        entry => stack%first
        do while (associated (entry))
            if (entry%get_id () == id) then
                ptr => entry%process_t
                return
            end if
            entry => entry%next
        end do
    end function process_stack_get_process_ptr
```

### 15.6.5 Auxiliary stuff

Write a separator line.

```
<Process stacks: procedures>+≡
    subroutine write_separator (u)
        integer, intent(in) :: u
        write (u, "(A)") repeat ("-", 72)
    end subroutine write_separator

    subroutine write_separator_double (u)
        integer, intent(in) :: u
        write (u, "(A)") repeat ("=", 72)
    end subroutine write_separator_double
```

### 15.6.6 Unit tests

```
<Process stacks: public>+≡
    public :: process_stacks_test

<Process stacks: tests>≡
    subroutine process_stacks_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Process stacks: execute tests>
    end subroutine process_stacks_test
```

## Write an empty process stack

The most trivial test is to write an uninitialized process stack.

```
<Process stacks: execute tests>≡
    call test (process_stacks_1, "process_stacks_1", &
        "write an empty process stack", &
        u, results)

<Process stacks: tests>+≡
    subroutine process_stacks_1 (u)
        integer, intent(in) :: u
        type(process_stack_t) :: stack

        write (u, "(A)")  "* Test output: process_stacks_1"
        write (u, "(A)")  "* Purpose: display an empty process stack"
        write (u, "(A)")

        call stack%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: process_stacks_1"

    end subroutine process_stacks_1
```

## Fill a process stack

Fill a process stack with two (identical) processes.

```
<Process stacks: execute tests>+≡
    call test (process_stacks_2, "process_stacks_2", &
        "fill a process stack", &
        u, results)

<Process stacks: tests>+≡
    subroutine process_stacks_2 (u)
        integer, intent(in) :: u
        type(process_stack_t) :: stack
        type(process_library_t), target :: lib
        type(string_t) :: libname
        type(string_t) :: procname
        type(string_t) :: run_id
        type(os_data_t) :: os_data
        type(process_entry_t), pointer :: process => null ()

        write (u, "(A)")  "* Test output: process_stacks_2"
        write (u, "(A)")  "* Purpose: fill a process stack"
        write (u, "(A)")

        write (u, "(A)")  "* Build, initialize and store two test processes"
        write (u, "(A)")

        libname = "process_stacks2"
        procname = libname
        call os_data_init (os_data)
        call prc_test_create_library (libname, lib)
```

```

call syntax_model_file_init ()

allocate (process)
run_id = "run1"
call process%init (procname, run_id, lib, os_data)
call stack%push (process)

allocate (process)
run_id = "run2"
call process%init (procname, run_id, lib, os_data)
call stack%push (process)

call stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call stack%final ()

call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_stacks_2"

end subroutine process_stacks_2

```

## 15.7 Complete Events

This module combines hard processes with decay chains, shower, and hadronization (not implemented yet) to complete events. It also manages the input and output of event records in various formats.

```

<events.f90>≡
  <File header>

  module events

    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use unit_tests
    use os_interface

    use ifiles
    use lexers
    use parser

    use subevents
    use variables
    use expressions
    use models

```

```

    use state_matrices
    use particles
    use interactions
    use subevt_expr
    use processes

```

*⟨Standard module head⟩*

*⟨Events: public⟩*

*⟨Events: types⟩*

*⟨Events: interfaces⟩*

contains

*⟨Events: procedures⟩*

*⟨Events: tests⟩*

end module events

### 15.7.1 Event configuration

The parameters govern the transformation of an event to a particle set.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions that apply to the current event. The workspaces for evaluating those expressions are set up in the `event_expr_t` objects. Note that these are really pointers, so the actual nodes are not stored inside the event object.

*⟨Events: types⟩*≡

```

    type :: event_config_t
        logical :: unweighted = .false.
        integer :: factorization_mode = FM_IGNORE_HELICITY
        logical :: keep_correlations = .false.
        type(parse_node_t), pointer :: pn_selection => null ()
        type(parse_node_t), pointer :: pn_reweight => null ()
        type(parse_node_t), pointer :: pn_analysis => null ()
    contains
        ⟨Events: event config: TBP⟩
    end type event_config_t

```

Output.

*⟨Events: event config: TBP⟩*≡

```

    procedure :: write => event_config_write

```

*⟨Events: procedures⟩*≡

```

    subroutine event_config_write (object, unit, show_expressions)
        class(event_config_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: show_expressions
        integer :: u
        u = output_unit (unit)
    end subroutine event_config_write

```

```

write (u, "(3x,A,L1)") "Unweighted          = ", object%unweighted
write (u, "(3x,A)", advance="no") "Helicity handling = "
select case (object%factorization_mode)
case (FM_IGNORE_HELICITY)
    write (u, "(A)") "drop"
case (FM_SELECT_HELICITY)
    write (u, "(A)") "select"
case (FM_FACTOR_HELICITY)
    write (u, "(A)") "factorize"
end select
write (u, "(3x,A,L1)") "Keep correlations = ", object%keep_correlations
if (present (show_expressions)) then
    if (show_expressions) then
        if (associated (object%pn_selection)) then
            call write_separator (u)
            write (u, "(3x,A)") "Event selection expression:"
            call object%pn_selection%write (u)
        end if
        if (associated (object%pn_reweight)) then
            call write_separator (u)
            write (u, "(3x,A)") "Event reweighting expression:"
            call object%pn_reweight%write (u)
        end if
        if (associated (object%pn_analysis)) then
            call write_separator (u)
            write (u, "(3x,A)") "Analysis expression:"
            call object%pn_analysis%write (u)
        end if
    end if
end if
end if
end subroutine event_config_write

```

## 15.7.2 The event type

Each event has a single elementary process as its core. This process may be dressed by a shower, a decay chain etc. For this core we implement pointers to a process instance.

The actual (dressed) event record is implemented as a `particle_set` object.

All data that involve user-provided expressions (selection, reweighting, analysis) are handled by the `expr` subobject. In particular, evaluating the event-selection expression sets the `passed` flag.

*(Events: public)*≡

```
public :: event_t
```

*(Events: types)*+≡

```

type :: event_t
    type(event_config_t) :: config
    type(process_t), pointer :: process => null ()
    type(process_instance_t), pointer :: instance => null ()
    integer :: selected_i_mci = 0
    integer :: selected_i_term = 0
    integer :: selected_channel = 0

```

```

    logical :: is_complete = .false.
    logical :: particle_set_exists = .false.
    type(particle_set_t) :: particle_set
    logical :: sqme_is_known = .false.
    real(default) :: sqme = 0
    real(default) :: weight = 0
    type(event_expr_t) :: expr
    logical :: selection_evaluated = .false.
    logical :: passed = .false.
    real(default) :: reweight = 1
    logical :: analysis_flag = .false.
contains
  <Events: event: TBP>
end type event_t

```

Finalizer: needed for the particle set.

```

<Events: event: TBP>≡
  procedure :: final => event_final

<Events: procedures>+≡
  subroutine event_final (object)
    class(event_t), intent(inout) :: object
    call particle_set_final (object%particle_set)
  end subroutine event_final

```

Output.

```

<Events: event: TBP>+≡
  procedure :: write => event_write

<Events: procedures>+≡
  subroutine event_write (object, unit, verbose)
    class(event_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    logical :: verb
    integer :: u
    u = output_unit (unit)
    verb = .false.; if (present (verbose)) verb = verbose
    call write_separator_double (u)
    if (object%is_complete) then
      write (u, "(1x,A)") "Event"
    else
      write (u, "(1x,A)") "Event [incomplete]"
    end if
    call write_separator (u)
    call object%config%write (u)
    if (object%sqme_is_known) then
      call write_separator (u)
      write (u, "(3x,A,ES19.12)") "Squared matrix el. = ", object%sqme
      write (u, "(3x,A,ES19.12)") "Event weight          = ", object%weight
    end if
    if (object%selected_i_mci /= 0) then
      call write_separator (u)
      write (u, "(3x,A,I0)") "Selected MCI group = ", object%selected_i_mci
    end if
  end subroutine event_write

```

```

        write (u, "(3x,A,I0)") "Selected term      = ", object%selected_i_term
        write (u, "(3x,A,I0)") "Selected channel = ", object%selected_channel
    end if
    if (object%selection_evaluated) then
        call write_separator (u)
        write (u, "(3x,A,L1)") "Passed selection = ", object%passed
        if (object%passed) then
            write (u, "(3x,A,ES19.12)") &
                "Reweighting factor = ", object%reweight
            write (u, "(3x,A,L1)") &
                "Analysis flag      = ", object%analysis_flag
        end if
    end if
    if (associated (object%instance)) then
        if (verb) then
            call object%instance%write (u)
        else
            call object%instance%write_header (u)
        end if
        if (object%particle_set_exists) then
            call particle_set_write (object%particle_set, u)
            call write_separator_double (u)
        end if
        if (object%expr%subevt_filled) then
            call object%expr%write (u)
            call write_separator_double (u)
        end if
    else
        call write_separator_double (u)
        write (u, "(1x,A)") "Process instance: [undefined]"
        call write_separator_double (u)
    end if
end subroutine event_write

```

### 15.7.3 Initialization

Initialize: set configuration parameters, using a variable list. We do not call this `init`, because this method name will be used by a type extension.

```

<Events: event: TBP>+≡
    procedure :: basic_init => event_init

<Events: procedures>+≡
    subroutine event_init (event, var_list)
        class(event_t), intent(out) :: event
        type(var_list_t), intent(in) :: var_list
        event%config%unweighted = var_list_get_lval (var_list, &
            var_str ("?unweighted"))
    end subroutine event_init

```

We link the event to an existing process instance. This includes the variable list, which is linked to the process variable list. Note that this is not necessarily identical to the variable list used for event initialization.

The variable list will contain pointers to `event` subobjects, therefore the `target` attribute.

```

<Events: event: TBP>+≡
  procedure :: connect => event_connect

<Events: procedures>+≡
  subroutine event_connect (event, process_instance)
    class(event_t), intent(inout), target :: event
    type(process_instance_t), intent(in), target :: process_instance
    event%process => process_instance%process
    event%instance => process_instance
    call event%expr%setup_vars (event%process%get_sqrts ())
    call event%expr%link_var_list (event%process%get_var_list_ptr ())
  end subroutine event_connect

```

Set the parse nodes for the associated expressions, individually. The parse-node pointers may be null.

```

<Events: event: TBP>+≡
  procedure :: set_selection => event_set_selection
  procedure :: set_reweight => event_set_reweight
  procedure :: set_analysis => event_set_analysis

<Events: procedures>+≡
  subroutine event_set_selection (event, pn_selection)
    class(event_t), intent(inout) :: event
    type(parse_node_t), intent(in), pointer :: pn_selection
    event%config%pn_selection => pn_selection
  end subroutine event_set_selection

  subroutine event_set_reweight (event, pn_reweight)
    class(event_t), intent(inout) :: event
    type(parse_node_t), intent(in), pointer :: pn_reweight
    event%config%pn_reweight => pn_reweight
  end subroutine event_set_reweight

  subroutine event_set_analysis (event, pn_analysis)
    class(event_t), intent(inout) :: event
    type(parse_node_t), intent(in), pointer :: pn_analysis
    event%config%pn_analysis => pn_analysis
  end subroutine event_set_analysis

```

Create evaluation trees from the parse trees. The `target` attribute is required because the expressions contain pointers to event subobjects.

```

<Events: event: TBP>+≡
  procedure :: setup_expressions => event_setup_expressions

<Events: procedures>+≡
  subroutine event_setup_expressions (event)
    class(event_t), intent(inout), target :: event
    call event%expr%setup_selection (event%config%pn_selection)
    call event%expr%setup_analysis (event%config%pn_analysis)
    call event%expr%setup_reweight (event%config%pn_reweight)
  end subroutine event_setup_expressions

```



### 15.7.4 Evaluation

To fill the `particle_set`, i.e., the event record proper, we have to factorize the density matrix and either select a state or trace over unobserved quantum numbers. There are several parameters in the event configuration that control this.

Note: we have to provide two random numbers. This may be replaced by a call to a random-number generator.

```

<Events: event: TBP>+≡
  procedure :: factorize_process => event_factorize_process

<Events: procedures>+≡
  subroutine event_factorize_process (event, r)
    class(event_t), intent(inout) :: event
    real(default), dimension(2), intent(in) :: r
    integer :: i_term
    type(interaction_t), pointer :: int_matrix, int_flows
    integer, dimension(:), allocatable :: beam_index
    integer, dimension(:), allocatable :: in_index
    integer :: n_in
    event%particle_set_exists = .false.
    if (event%instance%is_complete_event ()) then
      call event%instance%select_i_term (i_term)
      event%selected_i_term = i_term
      int_matrix => event%instance%get_matrix_int_ptr (i_term)
      int_flows => event%instance%get_flows_int_ptr (i_term)
      n_in = event%process%get_n_in ()
      call particle_set_init (event%particle_set, event%particle_set_exists, &
        int_matrix, int_flows, event%config%factorization_mode, r, &
        event%config%keep_correlations, keep_virtual=.true.)
      allocate (beam_index (n_in))
      call event%instance%get_beam_index (i_term, beam_index)
      call particle_set_reset_status (event%particle_set, &
        beam_index, PRT_BEAM)
      allocate (in_index (n_in))
      call event%instance%get_in_index (i_term, in_index)
      call particle_set_reset_status (event%particle_set, &
        in_index, PRT_INCOMING)
      event%particle_set_exists = .true.
    end if
    event%is_complete = event%sqme_is_known .and. event%particle_set_exists
  end subroutine event_factorize_process

```

Evaluate the event-related expressions, given a valid `particle_set`.

```

<Events: event: TBP>+≡
  procedure :: evaluate_expressions => event_evaluate_expressions

<Events: procedures>+≡
  subroutine event_evaluate_expressions (event)
    class(event_t), intent(inout) :: event
    if (event%particle_set_exists) then
      call event%expr%fill_subvt (event%particle_set)
      call event%expr%evaluate &
        (event%passed, event%reweight, event%analysis_flag)
    end if
  end subroutine event_evaluate_expressions

```

```

        event%selection_evaluated = .true.
    else
        ! error?
    end if
end subroutine event_evaluate_expressions

```

### 15.7.5 Reset to empty state

Applying this, current event contents are marked as incomplete but are not deleted. In particular, the initialization is kept.

```

<Events: event: TBP>+≡
    procedure :: reset => event_reset

<Events: procedures>+≡
    subroutine event_reset (event)
        class(event_t), intent(inout) :: event
        event%selected_i_mci = 0
        event%selected_i_term = 0
        event%selected_channel = 0
        event%is_complete = .false.
        event%particle_set_exists = .false.
        event%sqme_is_known = .false.
        call event%expr%reset ()
        event%selection_evaluated = .false.
        event%passed = .false.
        event%analysis_flag = .false.
        if (associated (event%instance)) then
            call event%instance%reset (reset_mci = .true.)
        end if
    end subroutine event_reset

```

### 15.7.6 Generation

Assuming that we have a valid process associated to the event, we generate an event. We complete the event data, then factorize the spin density matrix and transfer it to the particle set.

Note: We should apply decays, shower, hadronization before the transfer. This is not (re-)implemented yet.

The explicit random number argument `r` should be generated by a random-number generator. It is used for the factorization algorithm.

```

<Events: event: TBP>+≡
    procedure :: generate => event_generate

<Events: procedures>+≡
    subroutine event_generate (event, i_mci, r)
        class(event_t), intent(inout) :: event
        integer, intent(in) :: i_mci
        real(default), dimension(2), intent(in) :: r
        call event%reset ()
        event%selected_i_mci = i_mci
        if (event%config%unweighted) then

```

```

        call event%process%generate_unweighted_event (event%instance, i_mci)
        call event%instance%evaluate_event_data ()
        call event%instance%normalize_weight ()
    else
        call event%process%generate_weighted_event (event%instance, i_mci)
        call event%instance%evaluate_event_data ()
    end if
    event%selected_channel = event%instance%get_channel ()
    call event%accept_sqme ()
    call event%factorize_process (r)
    call event%evaluate_expressions ()
end subroutine event_generate

```

Transfer the results of the process instance calculation to the event record header.

```

<Events: event: TBP>+≡
    procedure :: accept_sqme => event_accept_sqme

<Events: procedures>+≡
    subroutine event_accept_sqme (event)
        class(event_t), intent(inout) :: event
        if (associated (event%instance)) then
            if (event%instance%is_complete_event ()) then
                call event%set_sqme (event%instance%get_sqme (), &
                    event%instance%get_weight ())
            end if
        end if
    end subroutine event_accept_sqme

```

Copy squared matrix element and event weight into the event record.

```

<Events: event: TBP>+≡
    procedure :: set_sqme => event_set_sqme

<Events: procedures>+≡
    subroutine event_set_sqme (event, sqme, weight)
        class(event_t), intent(inout) :: event
        real(default), intent(in) :: sqme, weight
        event%sqme = sqme
        event%weight = weight
        event%sqme_is_known = .true.
        if (associated (event%instance)) then
            event%is_complete = event%instance%is_complete_event () &
                .and. event%sqme_is_known .and. event%particle_set_exists
        end if
    end subroutine event_set_sqme

```

### 15.7.7 Recovering an event

Select MC group, term, and integration channel.

```

<Events: event: TBP>+≡
    procedure :: select => event_select

```

```

<Events: procedures>+≡
  subroutine event_select (event, i_mci, i_term, channel)
    class(event_t), intent(inout) :: event
    integer, intent(in) :: i_mci, i_term, channel
    if (associated (event%instance)) then
      event%selected_i_mci = i_mci
      event%selected_i_term = i_term
      event%selected_channel = channel
    else
      call msg_bug ("Event: select term: process instance undefined")
    end if
  end subroutine event_select

```

Copy a particle set into the event record.

```

<Events: event: TBP>+≡
  procedure :: set_particle_set => event_set_particle_set

<Events: procedures>+≡
  subroutine event_set_particle_set (event, pset)
    class(event_t), intent(inout) :: event
    type(particle_set_t), intent(in) :: pset
    event%particle_set = pset
    call event%accept_particle_set ()
  end subroutine event_set_particle_set

```

Mark the particle set as existing, assuming that it has just been filled somehow.

```

<Events: event: TBP>+≡
  procedure :: accept_particle_set => event_accept_particle_set

<Events: procedures>+≡
  subroutine event_accept_particle_set (event)
    class(event_t), intent(inout) :: event
    event%particle_set_exists = .true.
  end subroutine event_accept_particle_set

```

Here we try to recover an event from the `particle_set` subobject and recalculate the structure functions and matrix elements. We have the appropriate `process` object and an initialized `process_instance` at hand, so beam and configuration data are known. From the `particle_set`, we get the momenta.

The quantum-number information may be incomplete, e.g., helicity information may be partial or absent. We recover the event just from the momentum configuration.

We do not transfer the matrix element from the process instance to the event record, as we do when generating an event. The event record may contain the matrix element as read from file, and the current calculation may use different parameters. We thus can compare old and new values.

The kinematical `weight` may also be known already. If yes, we pass it to the `evaluate_event_data` procedure. It should already be normalized. Otherwise, this routine will attempt to fetch it from the MCI record in the process instance, and we should normalize the result just as when generating events.

Evaluating event expressions must also be done separately.

```

<Events: event: TBP>+≡

```

```

    procedure :: recalculate => event_recalculate
  <Events: procedures>+≡
    subroutine event_recalculate (event, update_sqme)
      class(event_t), intent(inout) :: event
      logical, intent(in) :: update_sqme
      integer :: i_mci, i_term, channel
      if (event%particle_set_exists) then
        i_mci = event%selected_i_mci
        i_term = event%selected_i_term
        channel = event%selected_channel
        if (i_mci == 0 .or. i_term == 0 .or. channel == 0) then
          call msg_bug ("Event: recalculate: undefined selection parameters")
        end if
        call event%instance%choose_mci (i_mci)
        call event%instance%set_trace (event%particle_set, i_term)
        call event%instance%recover (channel, i_term, update_sqme)
        call event%process%recover_event (event%instance, i_term)
        if (event%sqme_is_known) then
          call event%instance%evaluate_event_data (weight = event%weight)
        else
          call event%instance%evaluate_event_data ()
          if (event%config%unweighted) then
            call event%instance%normalize_weight ()
          end if
        end if
        event%is_complete = event%instance%is_complete_event () &
          .and. event%sqme_is_known
      else
        call msg_bug ("Event: can't recalculate, particle set is undefined")
      end if
    end subroutine event_recalculate

```

### 15.7.8 Access content

Pointer to the associated process object.

```

  <Events: event: TBP>+≡
    procedure :: get_process_ptr => event_get_process_ptr

  <Events: procedures>+≡
    function event_get_process_ptr (event) result (ptr)
      class(event_t), intent(in) :: event
      type(process_t), pointer :: ptr
      ptr => event%process
    end function event_get_process_ptr

```

Read the event squared matrix element and weight.

```

  <Events: event: TBP>+≡
    procedure :: get_sqme => event_get_sqme
    procedure :: get_weight => event_get_weight

```

```

<Events: procedures>+≡
function event_get_sqme (event) result (sqme)
  class(event_t), intent(in) :: event
  real(default) :: sqme
  if (event%sqme_is_known) then
    sqme = event%sqme
  else
    call msg_bug ("Event: attempt to extract undefined sqme")
  end if
end function event_get_sqme

function event_get_weight (event) result (weight)
  class(event_t), intent(in) :: event
  real(default) :: weight
  if (event%sqme_is_known) then
    weight = event%weight
  else
    call msg_bug ("Event: attempt to extract undefined weight")
  end if
end function event_get_weight

```

Read the squared ME value as stored in the process instance. This may differ from the event sqme if the process instance has been recalculated.

```

<Events: event: TBP>+≡
procedure :: get_sqme_process => event_get_sqme_process

<Events: procedures>+≡
function event_get_sqme_process (event) result (sqme)
  class(event_t), intent(in) :: event
  real(default) :: sqme
  if (associated (event%instance)) then
    sqme = event%instance%get_sqme ()
  else
    call msg_bug ("Event: attempt to get sqme &
    &from undefined process instance")
  end if
end function event_get_sqme_process

```

Tell whether the particle set is defined:

```

<Events: event: TBP>+≡
procedure :: has_particle_set => event_has_particle_set

<Events: procedures>+≡
function event_has_particle_set (event) result (flag)
  class(event_t), intent(in) :: event
  logical :: flag
  flag = event%particle_set_exists
end function event_has_particle_set

```

Pointer to the particle set.

```

<Events: event: TBP>+≡
procedure :: get_particle_set_ptr => event_get_particle_set_ptr

```

```

<Events: procedures>+≡
function event_get_particle_set_ptr (event) result (ptr)
  class(event_t), intent(in), target :: event
  type(particle_set_t), pointer :: ptr
  ptr => event%particle_set
end function event_get_particle_set_ptr

```

Return the current values of indices: the MCI group of components, the term index (different terms corresponding, potentially, to different effective kinematics), and the MC integration channel. The `i_mci` call is delegated to the current process instance.

```

<Events: event: TBP>+≡
procedure :: get_i_mci => event_get_i_mci
procedure :: get_i_term => event_get_i_term
procedure :: get_channel => event_get_channel

<Events: procedures>+≡
function event_get_i_mci (event) result (i_mci)
  class(event_t), intent(in) :: event
  integer :: i_mci
  i_mci = event%selected_i_mci
end function event_get_i_mci

function event_get_i_term (event) result (i_term)
  class(event_t), intent(in) :: event
  integer :: i_term
  i_term = event%selected_i_term
end function event_get_i_term

function event_get_channel (event) result (channel)
  class(event_t), intent(in) :: event
  integer :: channel
  channel = event%selected_channel
end function event_get_channel

```

Return data used by external event formats.

```

<Events: event: TBP>+≡
procedure :: get_fac_scale => event_get_fac_scale
procedure :: get_alpha_s => event_get_alpha_s

<Events: procedures>+≡
function event_get_fac_scale (event) result (fac_scale)
  class(event_t), intent(in) :: event
  real(default) :: fac_scale
  fac_scale = event%instance%get_fac_scale (event%selected_i_term)
end function event_get_fac_scale

function event_get_alpha_s (event) result (alpha_s)
  class(event_t), intent(in) :: event
  real(default) :: alpha_s
  alpha_s = event%instance%get_alpha_s (event%selected_i_term)
end function event_get_alpha_s

```

### 15.7.9 Auxiliary stuff

Write a separator line.

```
<Events: procedures>+≡
  subroutine write_separator (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat (" ", 72)
  end subroutine write_separator

  subroutine write_separator_double (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("=", 72)
  end subroutine write_separator_double
```

Eliminate numerical noise in the `subevt` expression.

```
<Events: public>+≡
  public :: pacify

<Events: interfaces>≡
  interface pacify
    module procedure pacify_event
  end interface pacify

<Events: procedures>+≡
  subroutine pacify_event (event)
    class(event_t), intent(inout) :: event
    if (event%particle_set_exists) call pacify (event%particle_set)
    if (event%expr%subevt_filled) call pacify (event%expr)
  end subroutine pacify_event
```

### 15.7.10 Unit tests

```
<Events: public>+≡
  public :: events_test

<Events: tests>≡
  subroutine events_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <Events: execute tests>
  end subroutine events_test
```

#### Empty event record

```
<Events: execute tests>≡
  call test (events_1, "events_1", &
    "empty event record", &
    u, results)
```



```

<Events: tests>+≡
subroutine events_1 (u)
  integer, intent(in) :: u
  type(event_t), target :: event

  write (u, "(A)")  "* Test output: events_1"
  write (u, "(A)")  "*   Purpose: display an empty event object"
  write (u, "(A)")

  call event%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: events_1"

end subroutine events_1

```

### Simple event

```

<Events: execute tests>+≡
call test (events_2, "events_2", &
  "generate event", &
  u, results)

<Events: tests>+≡
subroutine events_2 (u)
  integer, intent(in) :: u
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_instance_t), allocatable, target :: process_instance

  write (u, "(A)")  "* Test output: events_2"
  write (u, "(A)")  "*   Purpose: generate and display an event"
  write (u, "(A)")

  write (u, "(A)")  "* Generate test process event"

  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance)
  call process_instance%setup_event_data ()

  write (u, "(A)")
  write (u, "(A)")  "* Initialize event object"

  allocate (event)
  call event%connect (process_instance)

  write (u, "(A)")
  write (u, "(A)")  "* Generate test process event"

  call process%generate_weighted_event (process_instance, 1)

  write (u, "(A)")

```

```

write (u, "(A)")  "* Fill event object"
write (u, "(A)")

call event%generate (1, [0.4_default, 0.4_default])
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: events_2"

end subroutine events_2

```

## Event expressions

After generating an event, fill the subevt and evaluate expressions for selection, reweighting, and analysis.

```

<Events: execute tests>+≡
  call test (events_3, "events_3", &
    "expression evaluation", &
    u, results)

<Events: tests>+≡
  subroutine events_3 (u)
    integer, intent(in) :: u
    type(string_t) :: expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_selection, pt_reweight, pt_analysis
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: events_3"
    write (u, "(A)")  "* Purpose: generate an event and evaluate expressions"
    write (u, "(A)")

    write (u, "(A)")  "* Expression texts"
    write (u, "(A)")

    call syntax_pexpr_init ()

    expr_text = "all Pt > 100 [s]"
    write (u, "(A,A)")  "selection = ", char (expr_text)

```

```

call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_selection, stream, .true.)
call stream_final (stream)

expr_text = "1 + sqrts_hat / sqrts"
write (u, "(A,A)") "reweight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)

expr_text = "true"
write (u, "(A,A)") "analysis = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "/* Initialize test process event"

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance)
call process%set_var_list &
    (model_get_var_list_ptr (process%get_model_ptr ()))
call process_instance%setup_event_data ()

write (u, "(A)")
write (u, "(A)")  "/* Initialize event object and set expressions"

allocate (event)

call event%set_selection (parse_tree_get_root_ptr (pt_selection))
call event%set_reweight (parse_tree_get_root_ptr (pt_reweight))
call event%set_analysis (parse_tree_get_root_ptr (pt_analysis))

call event%connect (process_instance)
call event%setup_expressions ()

write (u, "(A)")
write (u, "(A)")  "/* Generate test process event"

call process%generate_weighted_event (process_instance, 1)

write (u, "(A)")
write (u, "(A)")  "/* Fill event object and evaluate expressions"
write (u, "(A)")

```

```

call event%generate (1, [0.4_default, 0.4_default])
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: events_3"

end subroutine events_3

```

### Recovering an event

Generate an event and store the particle set. Then reset the event record, recall the particle set, and recover the event from that.

```

<Events: execute tests>+≡
  call test (events_4, "events_4", &
    "recover event", &
    u, results)

<Events: tests>+≡
  subroutine events_4 (u)
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: particle_set

    write (u, "(A)")  "* Test output: events_4"
    write (u, "(A)")  "* Purpose: generate and recover an event"
    write (u, "(A)")

    write (u, "(A)")  "* Generate test process event and save particle set"
    write (u, "(A)")

    allocate (process)
    allocate (process_instance)
    call prepare_test_process (process, process_instance)
    call process_instance%setup_event_data ()

    allocate (event)
    call event%connect (process_instance)

    call event%generate (1, [0.4_default, 0.4_default])

```

```

call event%write (u)

particle_set = event%particle_set

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance)
call process_instance%setup_event_data ()

allocate (event)
call event%connect (process_instance)

call event%select (1, 1, 1)
call event%set_particle_set (particle_set)
call event%recalculate (update_sqme = .true.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Transfer sqme and evaluate expressions"
write (u, "(A)")

call event%accept_sqme ()
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset contents"
write (u, "(A)")

call event%reset ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (particle_set)

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: events_4"

end subroutine events_4

```

## Partially Recovering an event

Generate an event and store the particle set. Then reset the event record, recall the particle set, and recover the event as far as possible without recomputing the squared matrix element.

```

<Events: execute tests>+≡
  call test (events_5, "events_5", &
    "partially recover event", &
    u, results)

<Events: tests>+≡
  subroutine events_5 (u)
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: particle_set
    real(default) :: sqme, weight

    write (u, "(A)")  "* Test output: events_5"
    write (u, "(A)")  "* Purpose: generate and recover an event"
    write (u, "(A)")

    write (u, "(A)")  "* Generate test process event and save particle set"
    write (u, "(A)")

    allocate (process)
    allocate (process_instance)
    call prepare_test_process (process, process_instance)
    call process_instance%setup_event_data ()

    allocate (event)
    call event%connect (process_instance)

    call event%generate (1, [0.4_default, 0.4_default])
    call event%write (u)

    particle_set = event%particle_set
    sqme = event%get_sqme ()
    weight = event%get_weight ()

    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

```

```

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance)
call process_instance%setup_event_data ()

allocate (event)
call event%connect (process_instance)

call event%select (1, 1, 1)
call event%set_particle_set (particle_set)
call event%recalculate (update_sqme = .false.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually set sqme and evaluate expressions"
write (u, "(A)")

call event%set_sqme (sqme, weight)
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (particle_set)

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: events_5"

end subroutine events_5

```

## Chapter 16

# Matrix Element Implementations

In this chapter, we define concrete implementations for the workflow, from matrix-element code generation via matrix-element access to integration and event generation.

### 16.1 O’MEGA Interface

The standard method for process computation with WHIZARD is the O’MEGA matrix element generator.

This section implements the interface to the code generator (via the makefile) and the driver for the features provided by the O’MEGA matrix element.

```
<prc_omega.f90>≡  
  <File header>  
  
  module prc_omega  
  
    use iso_c_binding !NODEP!  
    use kinds !NODEP!  
    <Use file utils>  
    <Use strings>  
    use limits, only: TAB !NODEP!  
    use diagnostics !NODEP!  
    use unit_tests  
    use os_interface  
    use lorentz !NODEP!  
    use sm_qcd  
    use interactions  
    use variables  
    use models  
  
    use process_constants  
    use prclib_interfaces  
    use prc_core_def  
    use particle_specifiers  
    use process_libraries
```



```

    use prc_core

    <Standard module head>

    <Omega interface: public>

    <Omega interface: types>

    <Omega interface: interfaces>

    contains

    <Omega interface: procedures>

    <Omega interface: tests>

    end module prc_omega

```

### 16.1.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

```

<Omega interface: public>≡
    public :: omega_def_t

<Omega interface: types>≡
    type, extends (prc_core_def_t) :: omega_def_t
    contains
    <Omega interface: omega def: TBP>
    end type omega_def_t

<Omega interface: omega def: TBP>≡
    procedure, nopass :: type_string => omega_def_type_string

<Omega interface: procedures>≡
    function omega_def_type_string () result (string)
        type(string_t) :: string
        string = "omega"
    end function omega_def_type_string

```

Initialization: allocate the writer for the O'MEGA matrix element. Also set any data for this process that the writer needs.

```

<Omega interface: omega def: TBP>+≡
    procedure :: init => omega_def_init

<Omega interface: procedures>+≡
    subroutine omega_def_init (object, model_name, prt_in, prt_out, &
        restrictions, openmp_support, report_progress, extra_options)
        class(omega_def_t), intent(out) :: object
        type(string_t), intent(in) :: model_name
        type(string_t), dimension(:), intent(in) :: prt_in
        type(string_t), dimension(:), intent(in) :: prt_out
        type(string_t), intent(in), optional :: restrictions
        logical, intent(in), optional :: openmp_support
    end subroutine omega_def_init

```

```

logical, intent(in), optional :: report_progress
type(string_t), intent(in), optional :: extra_options
type(string_t) :: process_string
integer :: n_in
allocate (omega_writer_t :: object%writer)
select type (writer => object%writer)
type is (omega_writer_t)
    call writer%init (model_name, prt_in, prt_out, &
        restrictions, openmp_support, report_progress, extra_options)
end select
end subroutine omega_def_init

```

Write/read process- and method-specific data.

```

<Omega interface: omega def: TBP>+≡
    procedure :: write => omega_def_write

<Omega interface: procedures>+≡
    subroutine omega_def_write (object, unit)
        class(omega_def_t), intent(in) :: object
        integer, intent(in) :: unit
        select type (writer => object%writer)
        type is (omega_writer_t)
            call writer%write (unit)
        end select
    end subroutine omega_def_write

```

```

<Omega interface: omega def: TBP>+≡
    procedure :: read => omega_def_read

<Omega interface: procedures>+≡
    subroutine omega_def_read (object, unit)
        class(omega_def_t), intent(out) :: object
        integer, intent(in) :: unit
        call msg_bug ("O'Mega process definition: input not supported yet")
    end subroutine omega_def_read

```

Allocate the driver for O'MEGAmatrix elements.

```

<Omega interface: omega def: TBP>+≡
    procedure :: allocate_driver => omega_def_allocate_driver

<Omega interface: procedures>+≡
    subroutine omega_def_allocate_driver (object, driver, basename)
        class(omega_def_t), intent(in) :: object
        class(prc_core_driver_t), intent(out), allocatable :: driver
        type(string_t), intent(in) :: basename
        allocate (omega_driver_t :: driver)
    end subroutine omega_def_allocate_driver

```

We need code:

```

<Omega interface: omega def: TBP>+≡
    procedure, nopass :: needs_code => omega_def_needs_code

```

```

<Omega interface: procedures>+≡
  function omega_def_needs_code () result (flag)
    logical :: flag
    flag = .true.
  end function omega_def_needs_code

```

These are the features that an O'MEGA matrix element provides.

```

<Omega interface: omega def: TBP>+≡
  procedure, nopass :: get_features => omega_def_get_features

<Omega interface: procedures>+≡
  subroutine omega_def_get_features (features)
    type(string_t), dimension(:), allocatable, intent(out) :: features
    allocate (features (6))
    features = [ &
      var_str ("init"), &
      var_str ("update_alpha_s"), &
      var_str ("reset_helicity_selection"), &
      var_str ("is_allowed"), &
      var_str ("new_event"), &
      var_str ("get_amplitude")]
  end subroutine omega_def_get_features

```

The interface of the specific features.

```

<Omega interface: interfaces>≡
  abstract interface
    subroutine init_t (par) bind(C)
      import
      real(c_default_float), dimension(*), intent(in) :: par
    end subroutine init_t
  end interface

  abstract interface
    subroutine update_alpha_s_t (alpha_s) bind(C)
      import
      real(c_default_float), intent(in) :: alpha_s
    end subroutine update_alpha_s_t
  end interface

  abstract interface
    subroutine reset_helicity_selection_t (threshold, cutoff) bind(C)
      import
      real(c_default_float), intent(in) :: threshold
      integer(c_int), intent(in) :: cutoff
    end subroutine reset_helicity_selection_t
  end interface

  abstract interface
    subroutine is_allowed_t (flv, hel, col, flag) bind(C)
      import
      integer(c_int), intent(in) :: flv, hel, col
      logical(c_bool), intent(out) :: flag
    end subroutine is_allowed_t

```

```

end interface

abstract interface
  subroutine new_event_t (p) bind(C)
    import
    real(c_default_float), dimension(0:3,*), intent(in) :: p
  end subroutine new_event_t
end interface

abstract interface
  subroutine get_amplitude_t (flv, hel, col, amp) bind(C)
    import
    integer(c_int), intent(in) :: flv, hel, col
    complex(c_default_complex), intent(out):: amp
  end subroutine get_amplitude_t
end interface

```

Connect the O'MEGA features with the process driver.

```

<Omega interface: omega def: TBP>+≡
  procedure :: connect => omega_def_connect

<Omega interface: procedures>+≡
  subroutine omega_def_connect (def, lib_driver, i, proc_driver)
    class(omega_def_t), intent(in) :: def
    type(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    class(prc_core_driver_t), intent(inout) :: proc_driver
    integer(c_int) :: pid, fid
    type(c_funptr) :: fptr
    select type (proc_driver)
    type is (omega_driver_t)
      pid = i
      fid = 1
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%init)
      fid = 2
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%update_alpha_s)
      fid = 3
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%reset_helicity_selection)
      fid = 4
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%is_allowed)
      fid = 5
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%new_event)
      fid = 6
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%get_amplitude)
    end select
  end subroutine omega_def_connect

```

### 16.1.2 The O'MEGA writer

The O'MEGA writer is responsible for inserting the appropriate lines in the make-file that call O'MEGA, and for writing interfaces and wrappers.

```

(Omega interface: types)+≡
    type, extends (prc_writer_f_module_t) :: omega_writer_t
        type(string_t) :: model_name
        type(string_t) :: process_mode
        type(string_t) :: process_string
        type(string_t) :: restrictions
        logical :: openmp_support = .false.
        logical :: report_progress = .false.
        type(string_t) :: extra_options
    contains
        (Omega interface: omega writer: TBP)
    end type omega_writer_t

```

The reported type is the same as for the omega\_def\_t type.

```

(Omega interface: omega writer: TBP)≡
    procedure, nopass :: type_name => omega_writer_type_name

(Omega interface: procedures)+≡
    function omega_writer_type_name () result (string)
        type(string_t) :: string
        string = "omega"
    end function omega_writer_type_name

```

Output. This is called by omega\_def\_write.

```

(Omega interface: omega writer: TBP)+≡
    procedure :: write => omega_writer_write

(Omega interface: procedures)+≡
    subroutine omega_writer_write (object, unit)
        class(omega_writer_t), intent(in) :: object
        integer, intent(in) :: unit
        write (unit, "(5x,A,A)") "Mode string      = ", &
            ''' // char (object%process_mode) // '''
        write (unit, "(5x,A,A)") "Process string = ", &
            ''' // char (object%process_string) // '''
        write (unit, "(5x,A,A)") "Restrictions  = ", &
            ''' // char (object%restrictions) // '''
        write (unit, "(5x,A,L1)") "OpenMP support = ", object%openmp_support
        write (unit, "(5x,A,L1)") "Report progress = ", object%report_progress
        write (unit, "(5x,A,A)") "Extra options  = ", &
            ''' // char (object%extra_options) // '''
    end subroutine omega_writer_write

```

Initialize with process data.

```

(Omega interface: omega writer: TBP)+≡
    procedure :: init => omega_writer_init

```

*<Omega interface: procedures>+≡*

```

subroutine omega_writer_init (writer, model_name, prt_in, prt_out, &
    restrictions, openmp_support, report_progress, extra_options)
    class(omega_writer_t), intent(out) :: writer
    type(string_t), intent(in) :: model_name
    type(string_t), dimension(:), intent(in) :: prt_in
    type(string_t), dimension(:), intent(in) :: prt_out
    type(string_t), intent(in), optional :: restrictions
    logical, intent(in), optional :: openmp_support
    logical, intent(in), optional :: report_progress
    type(string_t), intent(in), optional :: extra_options
    integer :: i
    writer%model_name = model_name
    if (present (restrictions)) then
        writer%restrictions = restrictions
    else
        writer%restrictions = ""
    end if
    if (present (openmp_support)) writer%openmp_support = openmp_support
    if (present (report_progress)) writer%report_progress = report_progress
    if (present (extra_options)) then
        writer%extra_options = " " // extra_options
    else
        writer%extra_options = ""
    end if
    select case (size (prt_in))
    case (1); writer%process_mode = " -decay"
    case (2); writer%process_mode = " -scatter"
    end select
    associate (s => writer%process_string)
        s = " '"
        do i = 1, size (prt_in)
            if (i > 1) s = s // " "
            s = s // prt_in(i)
        end do
        s = s // " ->"
        do i = 1, size (prt_out)
            s = s // " " // prt_out(i)
        end do
        s = s // "' "
    end associate
end subroutine omega_writer_init

```

The makefile implements the actual O'MEGA call.

*<Omega interface: omega writer: TBP>+≡*

```

procedure :: write_makefile_code => omega_write_makefile_code

```

*<Omega interface: procedures>+≡*

```

subroutine omega_write_makefile_code (writer, unit, id, os_data)
    class(omega_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    type(string_t) :: omega_binary, omega_path

```

```

type(string_t) :: restrictions_string
type(string_t) :: openmp_string
type(string_t) :: progress_string
omega_binary = "omega_" // writer%model_name // ".opt"
omega_path = os_data%whizard_omega_binpath // "/" // omega_binary
if (writer%restrictions /= "") then
    restrictions_string = " -cascade '" // writer%restrictions // "'"
else
    restrictions_string = ""
end if
if (writer%openmp_support) then
    openmp_string = " -target:openmp"
else
    openmp_string = ""
end if
if (writer%report_progress) then
    progress_string = " -fusion:progress"
else
    progress_string = ""
end if
write (unit, "(5A)") "SOURCES += ", char (id), ".f90"
write (unit, "(5A)") "OBJECTS += ", char (id), ".lo"
write (unit, "(5A)") char (id), ".f90:"
write (unit, "(99A)") TAB, char (omega_path), &
    " -o ", char (id), ".f90", &
    " -target:whizard", &
    " -target:parameter_module parameters-", char (writer%model_name), &
    " -target:module ", char (id), &
    " -target:md5sum '", writer%md5sum, "'", &
    char (openmp_string), &
    char (progress_string), &
    char (writer%process_mode), char (writer%process_string), &
    char (restrictions_string), &
    char (writer%extra_options)
write (unit, "(5A)") "clean-", char (id), ":"
write (unit, "(5A)") TAB, "rm -f ", char (id), ".f90"
write (unit, "(5A)") TAB, "rm -f ", char (id), ".mod"
write (unit, "(5A)") TAB, "rm -f ", char (id), ".lo"
write (unit, "(5A)") "CLEAN_SOURCES += ", char (id), ".f90"
write (unit, "(5A)") "CLEAN_OBJECTS += ", char (id), ".mod"
write (unit, "(5A)") "CLEAN_OBJECTS += ", char (id), ".lo"
write (unit, "(5A)") char (id), ".lo: ", char (id), ".f90"
write (unit, "(5A)") TAB, "$(LTF_COMPILE) $<"
end subroutine omega_write_makefile_code

```

The source is written by the makefile, so nothing to do here.

```

<Omega interface: omega writer: TBP>+≡
    procedure :: write_source_code => omega_write_source_code

<Omega interface: procedures>+≡
    subroutine omega_write_source_code (writer, id)
        class(omega_writer_t), intent(in) :: writer
        type(string_t), intent(in) :: id
    end subroutine omega_write_source_code

```

Return the name of a procedure that implements a given feature, as it is provided by the external matrix-element code. O'MEGA names some procedures differently, therefore we translate here and override the binding of the base type.

```

(Omega interface: omega writer: TBP)+≡
  procedure, nopass :: get_procname => omega_writer_get_procname

(Omega interface: procedures)+≡
  function omega_writer_get_procname (feature) result (name)
    type(string_t) :: name
    type(string_t), intent(in) :: feature
    select case (char (feature))
      case ("n_in");   name = "number_particles_in"
      case ("n_out");  name = "number_particles_out"
      case ("n_flv");  name = "number_flavor_states"
      case ("n_hel");  name = "number_spin_states"
      case ("n_col");  name = "number_color_flows"
      case ("n_cin");  name = "number_color_indices"
      case ("n_cf");   name = "number_color_factors"
      case ("flv_state"); name = "flavor_states"
      case ("hel_state"); name = "spin_states"
      case ("col_state"); name = "color_flows"
      case default
        name = feature
    end select
  end function omega_writer_get_procname

```

The interfaces for the O'MEGA-specific features.

```

(Omega interface: omega writer: TBP)+≡
  procedure :: write_interface => omega_write_interface

(Omega interface: procedures)+≡
  subroutine omega_write_interface (writer, unit, id, feature)
    class(omega_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(string_t), intent(in) :: feature
    type(string_t) :: name
    name = writer%get_c_procname (id, feature)
    write (unit, "(2x,9A)" "interface"
    select case (char (feature))
      case ("init")
        write (unit, "(5x,9A)" "subroutine ", char (name), " (par) bind(C)"
        write (unit, "(7x,9A)" "import"
        write (unit, "(7x,9A)" "real(c_default_float), dimension(*), &
          &intent(in) :: par"
        write (unit, "(5x,9A)" "end subroutine ", char (name)
      case ("update_alpha_s")
        write (unit, "(5x,9A)" "subroutine ", char (name), " (alpha_s) bind(C)"
        write (unit, "(7x,9A)" "import"
        write (unit, "(7x,9A)" "real(c_default_float), intent(in) :: alpha_s"
        write (unit, "(5x,9A)" "end subroutine ", char (name)
      case ("reset_helicity_selection")
        write (unit, "(5x,9A)" "subroutine ", char (name), " &

```



```

        &(threshold, cutoff) bind(C)"
write (unit, "(7x,9A)") "import"
write (unit, "(7x,9A)") "real(c_default_float), intent(in) :: threshold"
write (unit, "(7x,9A)") "integer(c_int), intent(in) :: cutoff"
write (unit, "(5x,9A)") "end subroutine ", char (name)
case ("is_allowed")
write (unit, "(5x,9A)") "subroutine ", char (name), " &
&(flv, hel, col, flag) bind(C)"
write (unit, "(7x,9A)") "import"
write (unit, "(7x,9A)") "integer(c_int), intent(in) :: flv, hel, col"
write (unit, "(7x,9A)") "logical(c_bool), intent(out) :: flag"
write (unit, "(5x,9A)") "end subroutine ", char (name)
case ("new_event")
write (unit, "(5x,9A)") "subroutine ", char (name), " (p) bind(C)"
write (unit, "(7x,9A)") "import"
write (unit, "(7x,9A)") "real(c_default_float), dimension(0:3,*), &
&intent(in) :: p"
write (unit, "(5x,9A)") "end subroutine ", char (name)
case ("get_amplitude")
write (unit, "(5x,9A)") "subroutine ", char (name), " &
&(flv, hel, col, amp) bind(C)"
write (unit, "(7x,9A)") "import"
write (unit, "(7x,9A)") "integer(c_int), intent(in) :: flv, hel, col"
write (unit, "(7x,9A)") "complex(c_default_complex), intent(out) &
&:: amp"
write (unit, "(5x,9A)") "end subroutine ", char (name)
end select
write (unit, "(2x,9A)") "end interface"
end subroutine omega_write_interface

```

The wrappers have to take into account conversion between C and Fortran data types.

NOTE: The case `c_default_float`  $\neq$  default is not yet covered.

*<Omega interface: omega writer: TBP>+≡*

```
procedure :: write_wrapper => omega_write_wrapper
```

*<Omega interface: procedures>+≡*

```

subroutine omega_write_wrapper (writer, unit, id, feature)
class(omega_writer_t), intent(in) :: writer
integer, intent(in) :: unit
type(string_t), intent(in) :: id, feature
type(string_t) :: name
name = writer%get_c_procname (id, feature)
write (unit, *)
select case (char (feature))
case ("init")
write (unit, "(9A)") "subroutine ", char (name), " (par) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"
write (unit, "(2x,9A)") "use ", char (id)
write (unit, "(2x,9A)") "real(c_default_float), dimension(*), &
&intent(in) :: par"
if (c_default_float == default) then
write (unit, "(2x,9A)") "call ", char (feature), " (par)"

```

```

end if
write (unit, "(9A)") "end subroutine ", char (name)
case ("update_alpha_s")
write (unit, "(9A)") "subroutine ", char (name), " (alpha_s) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"
write (unit, "(2x,9A)") "use ", char (id)
if (c_default_float == default) then
write (unit, "(2x,9A)") "real(c_default_float), intent(in) &
&:: alpha_s"
write (unit, "(2x,9A)") "call ", char (feature), " (alpha_s)"
end if
write (unit, "(9A)") "end subroutine ", char (name)
case ("reset_helicity_selection")
write (unit, "(9A)") "subroutine ", char (name), &
" (threshold, cutoff) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"
write (unit, "(2x,9A)") "use ", char (id)
if (c_default_float == default) then
write (unit, "(2x,9A)") "real(c_default_float), intent(in) &
&:: threshold"
write (unit, "(2x,9A)") "integer(c_int), intent(in) :: cutoff"
write (unit, "(2x,9A)") "call ", char (feature), &
" (threshold, int (cutoff))"
end if
write (unit, "(9A)") "end subroutine ", char (name)
case ("is_allowed")
write (unit, "(9A)") "subroutine ", char (name), &
" (flv, hel, col, flag) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"
write (unit, "(2x,9A)") "use ", char (id)
write (unit, "(2x,9A)") "integer(c_int), intent(in) :: flv, hel, col"
write (unit, "(2x,9A)") "logical(c_bool), intent(out) :: flag"
write (unit, "(2x,9A)") "flag = ", char (feature), &
" (int (flv), int (hel), int (col))"
write (unit, "(9A)") "end subroutine ", char (name)
case ("new_event")
write (unit, "(9A)") "subroutine ", char (name), " (p) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"
write (unit, "(2x,9A)") "use ", char (id)
if (c_default_float == default) then
write (unit, "(2x,9A)") "real(c_default_float), dimension(0:3,*), &
&intent(in) :: p"
write (unit, "(2x,9A)") "call ", char (feature), " (p)"
end if
write (unit, "(9A)") "end subroutine ", char (name)
case ("get_amplitude")
write (unit, "(9A)") "subroutine ", char (name), &
" (flv, hel, col, amp) bind(C)"
write (unit, "(2x,9A)") "use iso_c_binding"
write (unit, "(2x,9A)") "use kinds"

```

```

write (unit, "(2x,9A)") "use ", char (id)
write (unit, "(2x,9A)") "integer(c_int), intent(in) :: flv, hel, col"
write (unit, "(2x,9A)") "complex(c_default_complex), intent(out) &
&:: amp"
write (unit, "(2x,9A)") "amp = ", char (feature), &
" (int (flv), int (hel), int (col))"
write (unit, "(9A)") "end subroutine ", char (name)
end select
end subroutine omega_write_wrapper

```

### 16.1.3 Driver

```

<Omega interface: types>+≡
type, extends (prc_core_driver_t) :: omega_driver_t
  procedure(init_t), nopass, pointer :: &
    init => null ()
  procedure(update_alpha_s_t), nopass, pointer :: &
    update_alpha_s => null ()
  procedure(reset_helicity_selection_t), nopass, pointer :: &
    reset_helicity_selection => null ()
  procedure(is_allowed_t), nopass, pointer :: &
    is_allowed => null ()
  procedure(new_event_t), nopass, pointer :: &
    new_event => null ()
  procedure(get_amplitude_t), nopass, pointer :: &
    get_amplitude => null ()
contains
  <Omega interface: omega driver: TBP>
end type omega_driver_t

```

The reported type is the same as for the `omega_def_t` type.

```

<Omega interface: omega driver: TBP>≡
  procedure, nopass :: type_name => omega_driver_type_name

<Omega interface: procedures>+≡
  function omega_driver_type_name () result (string)
    type(string_t) :: string
    string = "omega"
  end function omega_driver_type_name

```

### 16.1.4 High-level process definition

This procedure wraps the details filling a process-component definition entry as appropriate for an O'MEGA matrix element.

NOTE: For calling the `import_component` method, we must explicitly address the `process_def_t` parent object. The natural way to call the method of the extended type triggers a bug in gfortran 4.6. The string array arguments `prt_in` and `prt_out` become corrupted and cause a segfault.

```

<Omega interface: procedures>+≡
  subroutine omega_make_process_component (entry, component_index, &

```

```

        model_name, prt_in, prt_out, &
        restrictions, openmp_support, report_progress, extra_options)
class(process_def_entry_t), intent(inout) :: entry
integer, intent(in) :: component_index
type(string_t), intent(in) :: model_name
type(string_t), dimension(:), intent(in) :: prt_in
type(string_t), dimension(:), intent(in) :: prt_out
type(string_t), intent(in), optional :: restrictions
logical, intent(in), optional :: openmp_support
logical, intent(in), optional :: report_progress
type(string_t), intent(in), optional :: extra_options
class(prc_core_def_t), allocatable :: def
allocate (omega_def_t :: def)
select type (def)
type is (omega_def_t)
    call def%init (model_name, prt_in, prt_out, &
        restrictions, openmp_support, report_progress, extra_options)
end select
call entry%process_def_t%import_component (component_index, &
    n_out = size (prt_out), &
    prt_in = new_prt_spec (prt_in), &
    prt_out = new_prt_spec (prt_out), &
    method = var_str ("omega"), &
    variant = def)
end subroutine omega_make_process_component

```

### 16.1.5 The prc\_omega\_t wrapper

This is an instance of the generic `prc_core_t` object. It contains a pointer to the process definition (`omega_def_t`), a data component (`process_constants_t`), and the matrix-element driver (`omega_driver_t`).

```

<Omega interface: public>+≡
    public :: prc_omega_t

<Omega interface: types>+≡
    type, extends (prc_core_t) :: prc_omega_t
        real(default), dimension(:), allocatable :: par
        type(helicity_selection_t) :: helicity_selection
        type(qcd_t) :: qcd
    contains
        <Omega interface: prc omega: TBP>
    end type prc_omega_t

```

The workspace associated to a `prc_omega_t` object contains a single flag. The flag is used to suppress re-evaluating the matrix element for each quantum-number combination, after the first amplitude belonging to a given kinematics has been computed.

We can also store the value of a running coupling once it has been calculated for an event. The default value is negative, which indicates an undefined value in this context.

```

<Omega interface: types>+≡

```

```

type, extends (workspace_t) :: omega_state_t
  logical :: new_kinematics = .true.
  real(default) :: alpha_qcd = -1
contains
  procedure :: write => omega_state_write
end type omega_state_t

```

```

<Omega interface: procedures>+≡
  subroutine omega_state_write (object, unit)
    class(omega_state_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A,L1)") "0'Mega state: new kinematics = ", &
      object%new_kinematics
  end subroutine omega_state_write

```

Allocate the workspace with the above specific type.

```

<Omega interface: prc omega: TBP>≡
  procedure :: allocate_workspace => prc_omega_allocate_workspace

<Omega interface: procedures>+≡
  subroutine prc_omega_allocate_workspace (object, tmp)
    class(prc_omega_t), intent(in) :: object
    class(workspace_t), intent(inout), allocatable :: tmp
    allocate (omega_state_t :: tmp)
  end subroutine prc_omega_allocate_workspace

```

The following procedures are inherited from the base type as deferred, thus must be implemented. The corresponding unit tests are skipped here; the procedures are tested when called from the `processes` module.

Output: print just the ID of the associated matrix element. Then display any stored parameters and the helicity selection data. (The latter are printed only if active.)

```

<Omega interface: prc omega: TBP>+≡
  procedure :: write => prc_omega_write

<Omega interface: procedures>+≡
  subroutine prc_omega_write (object, unit)
    class(prc_omega_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(3x,A)", advance="no") "0'Mega process core:"
    if (object%data_known) then
      write (u, "(1x,A)") char (object%data%id)
    else
      write (u, "(1x,A)") "[undefined]"
    end if
    if (allocated (object%par)) then
      write (u, "(3x,A)") "Parameter array:"
      do i = 1, size (object%par)
        write (u, "(5x,I0,1x,ES17.10)") i, object%par(i)
      end do
    end if
  end subroutine prc_omega_write

```

```

        end do
    end if
    call object%helicity_selection%write (u)
    call object%qcd%write (u)
end subroutine prc_omega_write

```

Temporarily store the parameter array inside the `prc_omega` object, so we can use it later during the actual initialization. Also store threshold and cutoff for helicity selection.

```

<Omega interface: prc_omega: TBP>+=
    procedure :: set_parameters => prc_omega_set_parameters

<Omega interface: procedures>+=
    subroutine prc_omega_set_parameters &
        (prc_omega, model, helicity_selection, qcd)
        class(prc_omega_t), intent(inout) :: prc_omega
        type(model_t), intent(in), target, optional :: model
        type(helicity_selection_t), intent(in), optional :: helicity_selection
        type(qcd_t), intent(in), optional :: qcd
        if (present (model)) then
            if (allocated (prc_omega%par)) deallocate (prc_omega%par)
            call model_parameters_to_c_array (model, prc_omega%par)
        end if
        if (present (helicity_selection)) then
            prc_omega%helicity_selection = helicity_selection
        end if
        if (present (qcd)) then
            prc_omega%qcd = qcd
        end if
    end subroutine prc_omega_set_parameters

```

To fully initialize the process core, we perform base initialization, then initialize the external matrix element code.

This procedure overrides the `init` method of the base type, which we nevertheless can access via its binding `base_init`. When done, we have an allocated driver. The driver will call the `init` procedure for the external matrix element, and thus transfer the parameter set to where it finally belongs.

If requested, we initialize the helicity selection counter.

```

<Omega interface: prc_omega: TBP>+=
    procedure :: init => prc_omega_init

<Omega interface: procedures>+=
    subroutine prc_omega_init (object, def, lib, id, i_component)
        class(prc_omega_t), intent(inout) :: object
        class(prc_core_def_t), intent(in), target :: def
        type(process_library_t), intent(in), target :: lib
        type(string_t), intent(in) :: id
        integer, intent(in) :: i_component
        call object%base_init (def, lib, id, i_component)
        call object%activate_parameters ()
    end subroutine prc_omega_init

```

Activate the stored parameters by transferring them to the external matrix element. Also reset the helicity selection, if requested.

```

<Omega interface: prc omega: TBP>+≡
  procedure :: activate_parameters => prc_omega_activate_parameters

<Omega interface: procedures>+≡
  subroutine prc_omega_activate_parameters (object)
    class (prc_omega_t), intent(inout) :: object
    if (allocated (object%driver)) then
      if (allocated (object%par)) then
        select type (driver => object%driver)
          type is (omega_driver_t)
            if (associated (driver%init)) call driver%init (object%par)
          end select
        else
          call msg_bug ("prc_omega_activate: parameter set is not allocated")
        end if
        call object%reset_helicity_selection ()
      else
        call msg_bug ("prc_omega_activate: driver is not allocated")
      end if
    end subroutine prc_omega_activate_parameters

```

The O'MEGA process is an independent process which needs its own Monte Carlo parameter set for integration.

```

<Omega interface: prc omega: TBP>+≡
  procedure :: needs_mcset => prc_omega_needs_mcset

<Omega interface: procedures>+≡
  function prc_omega_needs_mcset (object) result (flag)
    class(prc_omega_t), intent(in) :: object
    logical :: flag
    flag = .true.
  end function prc_omega_needs_mcset

```

There is only one term for this tree-level process.

```

<Omega interface: prc omega: TBP>+≡
  procedure :: get_n_terms => prc_omega_get_n_terms

<Omega interface: procedures>+≡
  function prc_omega_get_n_terms (object) result (n)
    class(prc_omega_t), intent(in) :: object
    integer :: n
    n = 1
  end function prc_omega_get_n_terms

```

Tell whether a particular combination of flavor, helicity, color is allowed. Here we have to consult the matrix-element driver.

```

<Omega interface: prc omega: TBP>+≡
  procedure :: is_allowed => prc_omega_is_allowed

```

```

<Omega interface: procedures>+≡
function prc_omega_is_allowed (object, i_term, f, h, c) result (flag)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in) :: i_term, f, h, c
  logical :: flag
  logical(c_bool) :: cflag
  select type (driver => object%driver)
  type is (omega_driver_t)
    call driver%is_allowed (f, h, c, cflag)
    flag = cflag
  end select
end function prc_omega_is_allowed

```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

We reset the `new_kinematics` flag, so that the next call to `compute_amplitude` will evaluate the matrix element.

```

<Omega interface: prc omega: TBP>+≡
procedure :: compute_hard_kinematics => prc_omega_compute_hard_kinematics

<Omega interface: procedures>+≡
subroutine prc_omega_compute_hard_kinematics &
  (object, p_seed, i_term, int_hard, tmp)
  class(prc_omega_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(in) :: p_seed
  integer, intent(in) :: i_term
  type(interaction_t), intent(inout) :: int_hard
  class(workspace_t), intent(inout), allocatable :: tmp
  call interaction_set_momenta (int_hard, p_seed)
  if (allocated (tmp)) then
    select type (tmp)
    type is (omega_state_t); tmp%new_kinematics = .true.
    end select
  end if
end subroutine prc_omega_compute_hard_kinematics

```

This procedure is not called for `prc_omega_t`, just a placeholder.

```

<Omega interface: prc omega: TBP>+≡
procedure :: compute_eff_kinematics => prc_omega_compute_eff_kinematics

<Omega interface: procedures>+≡
subroutine prc_omega_compute_eff_kinematics &
  (object, i_term, int_hard, int_eff, tmp)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in) :: i_term
  type(interaction_t), intent(in) :: int_type
  type(interaction_t), intent(inout) :: int_eff
  class(workspace_t), intent(inout), allocatable :: tmp
end subroutine prc_omega_compute_eff_kinematics

```

Recover the momenta within the hard interaction. For tree-level processes, this is a trivial copy of the incoming seed and outgoing effective momenta. The



effective interaction is a pointer to the hard interaction, so the latter is ignored.

```

(Omega interface: prc omega: TBP)+≡
  procedure :: recover_kinematics => prc_omega_recover_kinematics

(Omega interface: tests)≡
  subroutine prc_omega_recover_kinematics &
    (object, p_seed, int_hard, int_eff, tmp)
    class(prc_omega_t), intent(in) :: object
    type(vector4_t), dimension(:), intent(inout) :: p_seed
    type(interaction_t), intent(inout) :: int_hard
    type(interaction_t), intent(inout) :: int_eff
    class(workspace_t), intent(inout), allocatable :: tmp
    integer :: n_in
    n_in = interaction_get_n_in (int_eff)
    call interaction_set_momenta (int_eff, p_seed(1:n_in), outgoing = .false.)
    p_seed(n_in+1:) = interaction_get_momenta (int_eff, outgoing = .true.)
  end subroutine prc_omega_recover_kinematics

```

Reset the helicity selection counters and start counting zero helicities. We assume that the `helicity_selection` object is allocated. Otherwise, reset and switch off helicity counting.

In the test routine, the driver is allocated but the driver methods are not. Therefore, guard against a disassociated method.

```

(Omega interface: prc omega: TBP)+≡
  procedure :: reset_helicity_selection => prc_omega_reset_helicity_selection

(Omega interface: procedures)+≡
  subroutine prc_omega_reset_helicity_selection (object)
    class(prc_omega_t), intent(inout) :: object
    select type (driver => object%driver)
    type is (omega_driver_t)
      if (associated (driver%reset_helicity_selection)) then
        if (object%helicity_selection%active) then
          call driver%reset_helicity_selection &
            (real (object%helicity_selection%threshold, &
              c_default_float), &
              int (object%helicity_selection%cutoff, c_int))
        else
          call driver%reset_helicity_selection &
            (0._c_default_float, 0_c_int)
        end if
      end if
    end select
  end subroutine prc_omega_reset_helicity_selection

```

Compute the amplitude. For the tree-level process, we can ignore the scale settings. The term index `j` is also irrelevant.

We first call `new_event` for the given momenta (which we must unpack), then retrieve the amplitude value for the given quantum numbers.

If the `tmp` status flag is present, we can make sure that we call `new_event` only once for a given kinematics. After the first call, we unset the `new_kinematics` flag.

```

(Omega interface: prc omega: TBP)+≡

```

```

procedure :: compute_amplitude => prc_omega_compute_amplitude
(Omega interface: procedures) +=
function prc_omega_compute_amplitude &
    (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
class(prc_omega_t), intent(in) :: object
integer, intent(in) :: j
type(vector4_t), dimension(:), intent(in) :: p
integer, intent(in) :: f, h, c
real(default), intent(in) :: fac_scale, ren_scale
class(workspace_t), intent(inout), allocatable, optional :: tmp
real(default) :: alpha_qcd
complex(default) :: amp
integer :: n_tot, i
real(c_default_float), dimension(:, :), allocatable :: parray
complex(c_default_complex) :: camp
logical :: new_event
select type (driver => object%driver)
type is (omega_driver_t)
    new_event = .true.
    if (present (tmp)) then
        if (allocated (tmp)) then
            select type (tmp)
            type is (omega_state_t)
                new_event = tmp%new_kinematics
                tmp%new_kinematics = .false.
            end select
        end if
    end if
    if (new_event) then
        if (allocated (object%qcd%alpha)) then
            alpha_qcd = object%qcd%alpha%get (fac_scale)
            call driver%update_alpha_s (alpha_qcd)
            if (present (tmp)) then
                if (allocated (tmp)) then
                    select type (tmp)
                    type is (omega_state_t)
                        tmp%alpha_qcd = alpha_qcd
                    end select
                end if
            end if
        end if
        n_tot = object%data%n_in + object%data%n_out
        allocate (parray (0:3, n_tot))
        forall (i = 1:n_tot) parray(:, i) = vector4_get_components (p(i))
        call driver%new_event (parray)
    end if
    if (object%is_allowed (1, f, h, c)) then
        call driver%get_amplitude &
            (int (f, c_int), int (h, c_int), int (c, c_int), camp)
        amp = camp
    else
        amp = 0
    end if
end select

```

```
end function prc_omega_compute_amplitude
```

After the amplitude has been computed, we may read off the current value of  $\alpha_s$ . This works only if  $\alpha_s$  varies, and if the workspace `tmp` is present which stores this value.

```
<Omega interface: prc_omega: TBP>+≡
  procedure :: get_alpha_s => prc_omega_get_alpha_s
<Omega interface: procedures>+≡
  function prc_omega_get_alpha_s (object, tmp) result (alpha)
    class(prc_omega_t), intent(in) :: object
    class(workspace_t), intent(in), allocatable :: tmp
    real(default) :: alpha
    alpha = -1
    if (allocated (object%qcd%alpha) .and. allocated (tmp)) then
      select type (tmp)
        type is (omega_state_t)
          alpha = tmp%alpha_qcd
        end select
      end if
    end function prc_omega_get_alpha_s
```

### 16.1.6 Test

This is the master for calling self-test procedures.

```
<Omega interface: public>+≡
  public :: prc_omega_test
<Omega interface: tests>+≡
  subroutine prc_omega_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <Omega interface: execute tests>
  end subroutine prc_omega_test
```

### Generate, compile and load a simple process matrix element

The process is  $e^+e^- \rightarrow \mu^+\mu^-$  for vanishing masses and  $e = 0.3$ . We initialize the process, build the library, and compute a particular matrix element for momenta of unit energy and right-angle scattering. The matrix element, as it happens, is equal to  $e^2$ . (Note that are no conversion factors applied, so this result is exact.)

For GNU `make`, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

```
<Omega interface: execute tests>≡
  call test (prc_omega_1, "prc_omega_1", &
    "build and load simple OMega process", &
    u, results)
```

```

(Omega interface: tests) +=
  subroutine prc_omega_1 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_def_t), allocatable :: def
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_constants_t) :: data
    class(prc_core_driver_t), allocatable :: driver
    integer, parameter :: cdf = c_default_float
    integer, parameter :: ci = c_int
    real(cdf), dimension(4) :: par
    real(cdf), dimension(0:3,4) :: p
    logical(c_bool) :: flag
    complex(c_default_complex) :: amp
    integer :: i

    write (u, "(A)")  "* Test output: prc_omega_1"
    write (u, "(A)")  "* Purpose: create a simple process with OMega"
    write (u, "(A)")  "*           build a library, link, load, and &
      &access the matrix element"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call lib%init (var_str ("omega1"))
    call os_data_init (os_data)

    model_name = "QED"
    allocate (prt_in (2), prt_out (2))
    prt_in = [var_str ("e+"), var_str ("e-")]
    prt_out = [var_str ("m+"), var_str ("m-")]

    allocate (omega_def_t :: def)
    select type (def)
    type is (omega_def_t)
      call def%init (model_name, prt_in, prt_out)
    end select
    allocate (entry)
    call entry%init (var_str ("omega1_a"), model_name = model_name, &
      n_in = 2, n_components = 1)
    call entry%import_component (1, n_out = size (prt_out), &
      prt_in = new_prt_spec (prt_in), &
      prt_out = new_prt_spec (prt_out), &
      method = var_str ("omega"), &
      variant = def)
    call lib%append (entry)

    write (u, "(A)")  "* Configure library"
    write (u, "(A)")
    call lib%configure ()

```

```

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
    lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
    lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of omega1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("omega1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID      = ", char (data%id)
write (u, "(1x,A,A)")  "model name        = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum          = '", data%md5sum, "'"
write (u, "(1x,A,L1)")  "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)")  "n_in            = ", data%n_in
write (u, "(1x,A,I0)")  "n_out           = ", data%n_out
write (u, "(1x,A,I0)")  "n_flv          = ", data%n_flv
write (u, "(1x,A,I0)")  "n_hel          = ", data%n_hel
write (u, "(1x,A,I0)")  "n_col          = ", data%n_col
write (u, "(1x,A,I0)")  "n_cin          = ", data%n_cin
write (u, "(1x,A,I0)")  "n_cf           = ", data%n_cf
write (u, "(1x,A,10(1x,I0))")  "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))")  "hel state =", data%hel_state(:,1)
do i = 2, 16
    write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))")  "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))")  "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))")  "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))")  "cf index =", data%cf_index

```

```

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega1_a and initialize:"
write (u, "(A)")

par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
write (u, "(2x,A,F6.4)")  "ee  = ", par(1)
write (u, "(2x,A,F6.4)")  "me  = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
    1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
    1.0_cdf, 0.0_cdf, 0.0_cdf, -1.0_cdf, &
    1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
    1.0_cdf, -1.0_cdf, 0.0_cdf, 0.0_cdf &
    ], [4,4])
do i = 1, 4
    write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

select type (driver)
type is (omega_driver_t)
    call driver%init (par)

    call driver%new_event (p)

    write (u, "(A)")
    write (u, "(A)")  "* Compute matrix element:"
    write (u, "(A)")

    call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
    write (u, "(1x,A,L1)")  "is_allowed (1, 6, 1) = ", flag

    call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
    write (u, "(1x,A,1x,E11.4)")  "|amp (1, 6, 1)| =", abs (amp)
end select

call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_omega_1"

end subroutine prc_omega_1

```

### Check prc\_omega\_t wrapper and options

The process is  $e^-e^+ \rightarrow e^-e^+$  for vanishing masses and  $e = 0.3$ . We build the library using the high-level procedure `omega_make_process_component` and

the “black box” `prc_omega_t` object. Two variants with different settings for restrictions and OpenMP.

For GNU make, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

```

(Omega interface: execute tests)+≡
    call test (prc_omega_2, "prc_omega_2", &
              "OMega option passing", &
              u, results)

(Omega interface: tests)+≡
    subroutine prc_omega_2 (u)
        integer, intent(in) :: u
        type(process_library_t), target :: lib
        type(process_def_entry_t), pointer :: entry
        type(os_data_t) :: os_data
        type(string_t) :: model_name
        type(model_t), pointer :: model => null ()
        type(var_list_t), pointer :: var_list => null ()
        type(string_t), dimension(:), allocatable :: prt_in, prt_out
        type(string_t) :: restrictions
        type(process_component_def_t), pointer :: config
        type(prc_omega_t) :: prc1, prc2
        type(process_constants_t) :: data
        integer, parameter :: cdf = c_default_float
        integer, parameter :: ci = c_int
        real(cdf), dimension(:), allocatable :: par
        real(cdf), dimension(0:3,4) :: p
        complex(c_default_complex) :: amp
        integer :: i, u_file
        logical :: exist

        write (u, "(A)")  "* Test output: prc_omega_2"
        write (u, "(A)")  "*   Purpose: create simple processes with OMega"
        write (u, "(A)")  "*               use the prc_omega wrapper for this"
        write (u, "(A)")  "*               and check OMega options"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize a process library with two entries, &
                           &different options."
        write (u, "(A)")  "* (1) e- e+ -> e- e+   &
                           &(all diagrams, no OpenMP, report progress)"
        write (u, "(A)")  "* (2) e- e+ -> e- e+   &
                           &(s-channel only, with OpenMP, report progress to file)"

        call lib%init (var_str ("omega2"))
        call os_data_init (os_data)
        call syntax_model_file_init ()

        model_name = "QED"
        call model_list_read_model &
            (var_str ("QED"), var_str ("QED.mdl"), os_data, model)
        var_list => model_get_var_list_ptr (model)

        allocate (prt_in (2), prt_out (2))

```

```

prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in
restrictions = "3+4~A"

allocate (entry)
call entry%init (var_str ("omega2_a"), &
    model, n_in = 2, n_components = 2)

call omega_make_process_component (entry, 1, &
    model_name, prt_in, prt_out, &
    report_progress=.true.)
call omega_make_process_component (entry, 2, &
    model_name, prt_in, prt_out, &
    restrictions=restrictions, openmp_support=.true., &
    extra_options=var_str ("-fusion:progress_file omega2.log"))

call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Remove left-over file"
write (u, "(A)")

call delete_file ("omega2.log")
inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure ()
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Check extra output of OMega"
write (u, "(A)")

inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
    lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
    lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega2_a and initialize:"
write (u, "(A)")

```



```

call var_list_set_real (var_list, var_str ("ee"), 0.3_default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("me"), 0._default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("mmu"), 0._default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("mtau"), 0._default, &
    is_known = .true.)
call model_parameters_to_c_array (model, par)

write (u, "(2x,A,F6.4)" "ee  = ", par(1)
write (u, "(2x,A,F6.4)" "me  = ", par(2)
write (u, "(2x,A,F6.4)" "mmu = ", par(3)
write (u, "(2x,A,F6.4)" "mtau = ", par(4)

call prc1%set_parameters (model)
call prc2%set_parameters (model)

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
    lib, var_str ("omega2_a"), 1)
call prc1%get_constants (data, 1)

write (u, "(1x,A,A)") "component ID      = ", &
    char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
    data%openmp_supported
write (u, "(1x,A,A,A)") "model name      = '", &
    char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i2:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 2)
call prc2%init (config%get_core_def_ptr (), &
    lib, var_str ("omega2_a"), 2)
call prc2%get_constants (data, 1)

write (u, "(1x,A,A)") "component ID      = ", &
    char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
    data%openmp_supported
write (u, "(1x,A,A,A)") "model name      = '", &
    char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

```

```

p = reshape ([ &
    1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
    1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
    1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
    1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
    ], [4,4])
do i = 1, 4
    write (u, "(2x,A,I0,A,4(1x,F7.4))") "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver => prc1%driver)
type is (omega_driver_t)
    call driver%new_event (p)
    call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
    write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
end select

select type (driver => prc2%driver)
type is (omega_driver_t)
    call driver%new_event (p)
    call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
    write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
end select

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
    1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
    1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
    1.0_cdf, sqrt(0.5_cdf), 0.0_cdf, sqrt(0.5_cdf), &
    1.0_cdf,-sqrt(0.5_cdf), 0.0_cdf,-sqrt(0.5_cdf) &
    ], [4,4])
do i = 1, 4
    write (u, "(2x,A,I0,A,4(1x,F7.4))") "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver => prc1%driver)
type is (omega_driver_t)
    call driver%new_event (p)
    call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
    write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
end select

```

```

select type (driver => prc2%driver)
type is (omega_driver_t)
  call driver%new_event (p)
  call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
  write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
end select

call lib%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_omega_2"

end subroutine prc_omega_2

```

### Check helicity selection

The process is  $e^-e^+ \rightarrow e^-e^+$  for vanishing masses. We call the matrix element several times to verify the switching off of irrelevant helicities.

```

<Omega interface: execute tests>+=
  call test (prc_omega_3, "prc_omega_3", &
    "helicity selection", &
    u, results)

<Omega interface: tests>+=
  subroutine prc_omega_3 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    type(model_t), pointer :: model => null ()
    type(var_list_t), pointer :: var_list => null ()
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_component_def_t), pointer :: config
    type(prc_omega_t) :: prc1
    type(process_constants_t) :: data
    integer, parameter :: cdf = c_default_float
    integer, parameter :: ci = c_int
    real(cdf), dimension(:), allocatable :: par
    real(cdf), dimension(0:3,4) :: p
    type(helicity_selection_t) :: helicity_selection
    integer :: i, h, u_file

    write (u, "(A)")  "* Test output: prc_omega_3"
    write (u, "(A)")  "*   Purpose: create simple process with OMega"
    write (u, "(A)")  "*               and check helicity selection"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library."
    write (u, "(A)")  "* (1) e- e+ -> e- e+   (all diagrams, no OpenMP)"

    call lib%init (var_str ("omega3"))

```

```

call os_data_init (os_data)
call syntax_model_file_init ()

model_name = "QED"
call model_list_read_model &
    (var_str ("QED"), var_str ("QED.mdl"), os_data, model)
var_list => model_get_var_list_ptr (model)

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in

allocate (entry)
call entry%init (var_str ("omega3_a"), &
    model, n_in = 2, n_components = 1)

call omega_make_process_component (entry, 1, &
    model_name, prt_in, prt_out)
call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure ()
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)") "is active" = ", &
    lib%is_active ()
write (u, "(1x,A,I0)") "n_processes" = ", &
    lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega3_a and initialize:"
write (u, "(A)")

call var_list_set_real (var_list, var_str ("ee"), 0.3_default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("me"), 0._default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("mmu"), 0._default, &
    is_known = .true.)
call var_list_set_real (var_list, var_str ("mtau"), 0._default, &
    is_known = .true.)
call model_parameters_to_c_array (model, par)

write (u, "(2x,A,F6.4)") "ee" = ", par(1)
write (u, "(2x,A,F6.4)") "me" = ", par(2)

```

```

write (u, "(2x,A,F6.4)") "mmu = ", par(3)
write (u, "(2x,A,F6.4)") "mtau = ", par(4)

call prc1%set_parameters (model, helicity_selection)

write (u, "(A)")
write (u, "(A)")  "* Helicity states of omega3_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega3_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
               lib, var_str ("omega3_a"), 1)
call prc1%get_constants (data, 1)

do i = 1, data%n_hel
    write (u, "(3x,I2,':',4(1x,I2))") i, data%hel_state(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Initially allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))") [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
    write (u, "(2x,L1)", advance = "no") prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection (cutoff = 4)"
write (u, "(A)")

helicity_selection%active = .true.
helicity_selection%threshold = 1e10_default
helicity_selection%cutoff = 4
call helicity_selection%write (u)

call prc1%set_parameters (model, helicity_selection)
call prc1%reset_helicity_selection ()

write (u, "(A)")
write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))") [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
    write (u, "(2x,L1)", advance = "no") prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"

```

```

write (u, "(A)")

p = reshape ([ &
    1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
    1.0_cdf, 0.0_cdf, 0.0_cdf, -1.0_cdf, &
    1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
    1.0_cdf, -1.0_cdf, 0.0_cdf, 0.0_cdf &
    ], [4,4])
do i = 1, 4
    write (u, "(2x,A,I0,A,4(1x,F7.4))") "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute scattering matrix 5 times"
write (u, "(A)")

write (u, "(4x,16(1x,I2))") [(h, h = 1, data%n_hel)]

select type (driver => prc1%driver)
type is (omega_driver_t)
    do i = 1, 5
        call driver%new_event (p)
        write (u, "(2x,I2)", advance = "no") i
        do h = 1, data%n_hel
            write (u, "(2x,L1)", advance = "no") prc1%is_allowed (1, 1, h, 1)
        end do
        write (u, "(A)")
    end do
end select

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection again"
write (u, "(A)")

call prc1%activate_parameters ()

write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))") [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
    write (u, "(2x,L1)", advance = "no") prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

call lib%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_omega_3"

end subroutine prc_omega_3

```

## QCD coupling

The process is  $u\bar{u} \rightarrow d\bar{d}$  for vanishing masses. We compute the amplitude for a fixed configuration once, then reset  $\alpha_s$ , then compute again.

For GNU make, makeflags is set to -j1. This eliminates a potential clash with a -j<n> flag if this test is called from a parallel make.

```
<Omega interface: execute tests>+≡
  call test (prc_omega_4, "prc_omega_4", &
    "update QCD alpha", &
    u, results)

<Omega interface: tests>+≡
  subroutine prc_omega_4 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_def_t), allocatable :: def
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_constants_t) :: data
    class(prc_core_driver_t), allocatable :: driver
    integer, parameter :: cdf = c_default_float
    integer, parameter :: ci = c_int
    real(cdf), dimension(6) :: par
    real(cdf), dimension(0:3,4) :: p
    logical(c_bool) :: flag
    complex(c_default_complex) :: amp
    integer :: i
    real(cdf) :: alpha_s

    write (u, "(A)")  "* Test output: prc_omega_4"
    write (u, "(A)")  "*   Purpose: create a QCD process with OMega"
    write (u, "(A)")  "*                   and check alpha_s dependence"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call lib%init (var_str ("prc_omega_4_lib"))
    call os_data_init (os_data)

    model_name = "QCD"
    allocate (prt_in (2), prt_out (2))
    prt_in = [var_str ("u"), var_str ("ubar")]
    prt_out = [var_str ("d"), var_str ("dbar")]

    allocate (omega_def_t :: def)
    select type (def)
    type is (omega_def_t)
      call def%init (model_name, prt_in, prt_out)
    end select
    allocate (entry)
```

```

call entry%init (var_str ("prc_omega_4_p"), model_name = model_name, &
  n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
  prt_in = new_prt_spec (prt_in), &
  prt_out = new_prt_spec (prt_out), &
  method = var_str ("omega"), &
  variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")
call lib%configure ()
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters:"
write (u, "(A)")

alpha_s = 0.1178_cdf

par = [alpha_s, 0._cdf, 0._cdf, 0._cdf, 173.1_cdf, 1.523_cdf]
write (u, "(2x,A,F8.4)")  "alpha_s = ", par(1)
write (u, "(2x,A,F8.4)")  "ms      = ", par(2)
write (u, "(2x,A,F8.4)")  "mc      = ", par(3)
write (u, "(2x,A,F8.4)")  "mb      = ", par(4)
write (u, "(2x,A,F8.4)")  "mtop    = ", par(5)
write (u, "(2x,A,F8.4)")  "wtop    = ", par(6)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
  100.0_cdf, 0.0_cdf, 0.0_cdf, 100.0_cdf, &
  100.0_cdf, 0.0_cdf, 0.0_cdf, -100.0_cdf, &
  100.0_cdf, 100.0_cdf, 0.0_cdf, 0.0_cdf, &
  100.0_cdf, -100.0_cdf, 0.0_cdf, 0.0_cdf &
], [4,4])
do i = 1, 4
  write (u, "(2x,A,I0,A,4(1x,F7.1))")  "p", i, " =", p(:,i)
end do

call lib%connect_process (var_str ("prc_omega_4_p"), 1, data, driver)

select type (driver)
type is (omega_driver_t)

```



```

call driver%init (par)

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

call driver%new_event (p)

call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

write (u, "(A)")
write (u, "(A)")  "* Double alpha_s and compute matrix element again:"
write (u, "(A)")

call driver%update_alpha_s (2 * alpha_s)
call driver%new_event (p)

call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
end select

call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_omega_4"

end subroutine prc_omega_4

```

## Amplitude and QCD coupling

The same process as before. Here, we initialize with a running  $\alpha_s$  coupling and compute twice with different scales. We use the high-level method `compute_amplitude`.

```

<Omega interface: execute tests>+≡
  call test (prc_omega_5, "prc_omega_5", &
    "running QCD alpha", &
    u, results)

<Omega interface: tests>+≡
subroutine prc_omega_5 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_component_def_t), pointer :: cdef_ptr
  class(prc_core_def_t), pointer :: def_ptr
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data

```

```

type(model_t), pointer :: model
type(string_t) :: model_name
type(string_t), dimension(:), allocatable :: prt_in, prt_out
type(qcd_t) :: qcd
class(prc_core_t), allocatable :: core
class(workspace_t), allocatable :: tmp
type(vector4_t), dimension(4) :: p
complex(default) :: amp
real(default) :: fac_scale
integer :: i

write (u, "(A)")  "* Test output: prc_omega_5"
write (u, "(A)")  "* Purpose: create a QCD process with OMega"
write (u, "(A)")  "*           and check alpha_s dependence"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry"
write (u, "(A)")
call lib%init (var_str ("prc_omega_5_lib"))
call os_data_init (os_data)

call syntax_model_file_init ()
call model_list_read_model (var_str ("QCD"), var_str ("QCD.mdl"), &
    os_data, model)
model_name = "QCD"

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("u"), var_str ("ubar")]
prt_out = [var_str ("d"), var_str ("dbar")]

allocate (omega_def_t :: def)
select type (def)
type is (omega_def_t)
    call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("prc_omega_5_p"), model_name = model_name, &
    n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
    prt_in = new_prt_spec (prt_in), &
    prt_out = new_prt_spec (prt_out), &
    method = var_str ("omega"), &
    variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")
call lib%configure ()
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API"

```

```

write (u, "(A)")

write (u, "(1x,A,L1)") "is active = ", lib%is_active ()

write (u, "(A)")
write (u, "(A)")  "* Set kinematics"
write (u, "(A)")

p(1) = vector4_moving (100._default, 100._default, 3)
p(2) = vector4_moving (100._default,-100._default, 3)
p(3) = vector4_moving (100._default, 100._default, 1)
p(4) = vector4_moving (100._default,-100._default, 1)
do i = 1, 4
    call vector4_write (p(i), u)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup QCD data"
write (u, "(A)")

allocate (alpha_qcd_from_scale_t :: qcd%alpha)

write (u, "(A)")  "* Setup process core"
write (u, "(A)")

allocate (prc_omega_t :: core)
cdef_ptr => lib%get_component_def_ptr (var_str ("prc_omega_5_p"), 1)
def_ptr => cdef_ptr%get_core_def_ptr ()

select type (core)
type is (prc_omega_t)
    call core%allocate_workspace (tmp)
    call core%set_parameters (model, qcd = qcd)
    call core%init (def_ptr, lib, var_str ("prc_omega_5_p"), 1)
    call core%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element"
write (u, "(A)")

fac_scale = 100
write (u, "(1x,A,F4.0)") "factorization scale = ", fac_scale

amp = core%compute_amplitude &
    (1, p, 1, 6, 1, fac_scale, 100._default)

write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

write (u, "(A)")
write (u, "(A)")  "* Modify factorization scale and &
    &compute matrix element again"
write (u, "(A)")

fac_scale = 200

```

```

write (u, "(1x,A,F4.0)") "factorization scale = ", fac_scale

amp = core%compute_amplitude &
      (1, p, 1, 6, 1, fac_scale, 100._default)

write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

end select

call lib%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_omega_5"

end subroutine prc_omega_5

```

# Chapter 17

## Event I/O

In this chapter we provide a generic handler for event I/O and the various implementations: raw, HepMC, Les Houches, etc.

### 17.1 Event Sample Data

We define a simple and transparent container for (meta)data that are associated with an event sample.

```
<eio_data.f90>≡  
<File header>  
  
module eio_data  
  
    use kinds !NODEP!  
    <Use file utils>  
    use unit_tests  
  
    <EIO data: public>  
  
    <EIO data: types>  
  
contains  
  
    <EIO data: procedures>  
  
    <EIO data: tests>  
  
end module eio_data
```

#### 17.1.1 Event Sample Data

These are data that apply to an event sample as a whole. They are given in an easily portable form (no fancy structure) and are used for initializing event formats.

There are two MD5 sums here. `md5sum_proc` depends only on the definition of the contributing processes. A sample with matching checksum can be

rescanned with modified model parameters, beam structure etc, to recalculate observables. `md5sum_config` includes all relevant data. Rescanning a sample with matching checksum will produce identical observables. (A third checksum might be added which depends on the event sample itself. This is not needed, so far.)

```

(EIO data: public)≡
    public :: event_sample_data_t

(EIO data: types)≡
    type :: event_sample_data_t
        character(32) :: md5sum_prc = ""
        character(32) :: md5sum_cfg = ""
        logical :: unweighted = .true.
        logical :: negative_weights = .false.
        integer :: n_beam = 0
        integer, dimension(2) :: pdg_beam = 0
        real(default), dimension(2) :: energy_beam = 0
        integer :: n_proc = 0
        integer, dimension(:), allocatable :: proc_num_id
        real(default), dimension(:), allocatable :: cross_section
        real(default), dimension(:), allocatable :: error
    contains
        (EIO data: event sample data: TBP)
    end type event_sample_data_t

```

Initialize: allocate for the number of processes

```

(EIO data: event sample data: TBP)≡
    procedure :: init => event_sample_data_init

(EIO data: procedures)≡
    subroutine event_sample_data_init (data, n_proc)
        class(event_sample_data_t), intent(out) :: data
        integer, intent(in) :: n_proc
        data%n_proc = n_proc
        allocate (data%proc_num_id (n_proc), source = 0)
        allocate (data%cross_section (n_proc), source = 0._default)
        allocate (data%error (n_proc), source = 0._default)
    end subroutine event_sample_data_init

```

Output.

```

(EIO data: event sample data: TBP)+≡
    procedure :: write => event_sample_data_write

(EIO data: procedures)+≡
    subroutine event_sample_data_write (data, unit)
        class(event_sample_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
        write (u, "(1x,A)") "Event sample properties:"
        write (u, "(3x,A,A,A)") "MD5 sum (proc) = ", data%md5sum_prc, ""
        write (u, "(3x,A,A,A)") "MD5 sum (config) = ", data%md5sum_cfg, ""
        write (u, "(3x,A,L1)") "unweighted = ", data%unweighted
        write (u, "(3x,A,L1)") "negative weights = ", data%negative_weights
    end subroutine event_sample_data_write

```

```

write (u, "(3x,A,I0)") "number of beams = ", data%n_beam
write (u, "(5x,A,2(1x,I19))") "PDG      = ", &
    data%pdg_beam(:data%n_beam)
write (u, "(5x,A,2(1x,ES19.12))") "Energy = ", &
    data%energy_beam(:data%n_beam)
write (u, "(3x,A,I0)") "num of processes = ", data%n_proc
do i = 1, data%n_proc
    write (u, "(3x,A,I0)") "Process #", data%proc_num_id (i)
    select case (data%n_beam)
    case (1)
        write (u, "(5x,A,ES19.12)") "Width = ", data%cross_section(i)
    case (2)
        write (u, "(5x,A,ES19.12)") "CSec  = ", data%cross_section(i)
    end select
    write (u, "(5x,A,ES19.12)") "Error = ", data%error(i)
end do
end subroutine event_sample_data_write

```

### 17.1.2 Unit tests

```

<EIO data: public>+≡
    public :: eio_data_test

<EIO data: tests>≡
    subroutine eio_data_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <EIO data: execute tests>
    end subroutine eio_data_test

```

### Event Sample Data

Print the contents of a sample data block.

```

<EIO data: execute tests>≡
    call test (eio_data_1, "eio_data_1", &
        "event sample data", &
        u, results)

<EIO data: tests>+≡
    subroutine eio_data_1 (u)
        integer, intent(in) :: u
        type(event_sample_data_t) :: data

        write (u, "(A)")  "* Test output: eio_data_1"
        write (u, "(A)")  "* Purpose: display event sample data"
        write (u, "(A)")

        write (u, "(A)")  "* Decay process, one component"
        write (u, "(A)")

        call data%init (1)
        data%n_beam = 1
    end subroutine eio_data_1

```

```

data%pdg_beam(1) = 25
data%energy_beam(1) = 125

data%proc_num_id = [42]
data%cross_section = [1.23e-4_default]
data%error = 5e-6_default

data%md5sum_prc = "abcdefghijklmnopabcdefghijklmnop"
data%md5sum_cfg = "12345678901234561234567890123456"

call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Scattering process, two components"
write (u, "(A)")

call data%init (2)
data%n_beam = 2
data%pdg_beam = [2212, -2212]
data%energy_beam = [8._default, 10._default]

data%proc_num_id = [12, 34]
data%cross_section = [100._default, 88._default]
data%error = [1._default, 0.1_default]

call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_data_1"

end subroutine eio_data_1

```

## 17.2 Abstract I/O Handler

This module defines an abstract object for event I/O and the associated methods.

There are `output` and `input` methods which write or read a single event from/to the I/O stream, respectively. The I/O stream itself may be a file, a common block, or an externally linked structure, depending on the concrete implementation.

A `write` method prints the current content of the implementation-dependent event record in human-readable form.

The `init_in/init_out` and `final` prepare and finalize the I/O stream, respectively. There is also a `switch_inout` method which turns an input stream into an output stream where events can be appended.

$\langle \text{eio\_base.f90} \rangle \equiv$   
 $\langle \text{File header} \rangle$

```

module eio_base

  use kinds !NODEP!

```



```

<Use file utils>
<Use strings>
  use diagnostics !NODEP!
  use unit_tests

  use lorentz !NODEP!
  use particles
  use beams
  use processes
  use events
  use eio_data

<Standard module head>

<EIO base: public>

<EIO base: types>

<EIO base: interfaces>

<EIO base: test types>

contains

<EIO base: procedures>

<EIO base: tests>

end module eio_base

```

### 17.2.1 Process pointer

For handling multiple processes, we need an array of process pointers. This requires a wrapper:

```

<EIO base: public>≡
  public :: process_ptr_t

<EIO base: types>≡
  type :: process_ptr_t
    type(process_t), pointer :: ptr => null ()
  end type process_ptr_t

```

### 17.2.2 Type

```

<EIO base: public>+≡
  public :: eio_t

<EIO base: types>+≡
  type, abstract :: eio_t
  contains
    <EIO base: eio base: TBP>
  end type eio_t

```

Write to screen. If possible, this should display the contents of the current event, i.e., the last one that was written or read.

```

<EIO base: eio base: TBP>≡
  procedure (eio_write), deferred :: write

<EIO base: interfaces>≡
  abstract interface
    subroutine eio_write (object, unit)
      import
      class(eio_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine eio_write
  end interface

```

Finalize. This should write/read footer data and close input/output channels.

```

<EIO base: eio base: TBP>+≡
  procedure (eio_final), deferred :: final

<EIO base: interfaces>+≡
  abstract interface
    subroutine eio_final (object)
      import
      class(eio_t), intent(inout) :: object
    end subroutine eio_final
  end interface

```

Initialize for output. We provide process pointers. This should open an event file if appropriate and write header data. Some methods may require event sample data.

```

<EIO base: eio base: TBP>+≡
  procedure (eio_init_out), deferred :: init_out

<EIO base: interfaces>+≡
  abstract interface
    subroutine eio_init_out (eio, sample, process_ptr, data, success)
      import
      class(eio_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
    end subroutine eio_init_out
  end interface

```

Initialize for input. We provide process names. This should open an event file if appropriate and read header data. The md5sum can be used to check the integrity of the configuration, if it provides a checksum to compare with.

```

<EIO base: eio base: TBP>+≡
  procedure (eio_init_in), deferred :: init_in

<EIO base: interfaces>+≡
  abstract interface
    subroutine eio_init_in (eio, sample, process_ptr, data, success)

```

```

import
class(eio_t), intent(inout) :: eio
type(string_t), intent(in) :: sample
type(process_ptr_t), dimension(:), intent(in) :: process_ptr
type(event_sample_data_t), intent(in), optional :: data
logical, intent(out), optional :: success
end subroutine eio_init_in
end interface

```

Re-initialize for output. This should change the status of any event file from input to output and position it for appending new events.

```

(EIO base: eio base: TBP)+≡
  procedure (eio_switch_inout), deferred :: switch_inout
(EIO base: interfaces)+≡
  abstract interface
    subroutine eio_switch_inout (eio, success)
      import
      class(eio_t), intent(inout) :: eio
      logical, intent(out), optional :: success
    end subroutine eio_switch_inout
  end interface

```

Output an event. All data can be taken from the `event` record. The index `i_prc` identifies the process among the processes that are contained in the current sample.

```

(EIO base: eio base: TBP)+≡
  procedure (eio_output), deferred :: output
(EIO base: interfaces)+≡
  abstract interface
    subroutine eio_output (eio, event, i_prc)
      import
      class(eio_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
    end subroutine eio_output
  end interface

```

Input an event. This should fill all event data that cannot be inferred from the associated process.

The input is broken down into two parts. First we read the `i_prc` index. So we know which process to expect in the subsequent event. If we have reached end of file, we also will know. Then, we read the event itself.

The parameter `iostat` is supposed to be set as the Fortran standard requires, negative for EOF and positive for error.

```

(EIO base: eio base: TBP)+≡
  procedure (eio_input_i_prc), deferred :: input_i_prc
  procedure (eio_input_event), deferred :: input_event
(EIO base: interfaces)+≡
  abstract interface
    subroutine eio_input_i_prc (eio, i_prc, iostat)

```

```

import
class(eio_t), intent(inout) :: eio
integer, intent(out) :: i_prc
integer, intent(out) :: iostat
end subroutine eio_input_i_prc
end interface

abstract interface
  subroutine eio_input_event (eio, event, iostat)
    import
    class(eio_t), intent(inout) :: eio
    type(event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
  end subroutine eio_input_event
end interface

```

### 17.2.3 Unit tests

```

<EIO base: public>+≡
  public :: eio_base_test

<EIO base: tests>≡
  subroutine eio_base_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <EIO base: execute tests>
  end subroutine eio_base_test

```

#### Test type for event I/O

The contents simulate the contents of an external file. We have the `sample` string as the file name, `sqrts` and the array of `process_ids` as metadata, and the array of momenta `event_p` as the list of events. The second index is the event index. The `event_i` component is the pointer to the current event, `event_n` is the total number of stored events.

```

<EIO base: test types>≡
  type, extends (eio_t) :: eio_test_t
    type(string_t) :: sample
    real(default) :: sqrts
    type(string_t) :: process_id
    integer :: event_n = 0
    integer :: event_i = 0
    integer :: i_prc = 0
    type(vector4_t), dimension(:,,:), allocatable :: event_p
  contains
  <EIO base: eio test: TBP>
  end type eio_test_t

```

Write to screen. Pretend that this is an actual event format.

```

<EIO base: eio test: TBP>≡
  procedure :: write => eio_test_write

```

```

<EIO base: procedures>≡
  subroutine eio_test_write (object, unit)
    class(eio_test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(1x,A)") "Test event stream"
    write (u, "(3x,A,A,A)") "Process ID = ", char (object%process_id), ""
    write (u, "(3x,A,ES19.12)") "sqrts      = ", object%sqrts
    if (object%event_i /= 0) then
      write (u, "(1x,A,I0,A)") "Event #", object%event_i, ":"
      do i = 1, size (object%event_p, 1)
        call vector4_write (object%event_p(i, object%event_i), u)
      end do
    end if
  end subroutine eio_test_write

```

Finalizer. For the test case, we just reset the event count, but keep the stored “events”. For the real implementations, the events would be stored on an external medium, so we would delete the object contents.

```

<EIO base: eio test: TBP>+≡
  procedure :: final => eio_test_final

<EIO base: procedures>+≡
  subroutine eio_test_final (object)
    class(eio_test_t), intent(inout) :: object
    object%event_i = 0
  end subroutine eio_test_final

```

Initialization: We store the process IDs and the energy from the beam-data object. We also allocate the momenta (i.e., the simulated event record) for a fixed maximum size of 10 events, 2 momenta each. There is only a single process.

```

<EIO base: eio test: TBP>+≡
  procedure :: init_out => eio_test_init_out

<EIO base: procedures>+≡
  subroutine eio_test_init_out (eio, sample, process_ptr, data, success)
    class(eio_test_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    type(beam_data_t), pointer :: beam_data
    eio%sample = sample
    eio%process_id = process_ptr(1)%ptr%get_id ()
    beam_data => process_ptr(1)%ptr%get_beam_data_ptr ()
    eio%sqrts = beam_data_get_sqrts (beam_data)
    eio%event_n = 0
    eio%event_i = 0
    allocate (eio%event_p (2, 10))
    if (present (success)) success = .true.
  end subroutine eio_test_init_out

```

Initialization for input. Nothing to do for the test type.

```

(EIO base: eio test: TBP)+≡
  procedure :: init_in => eio_test_init_in

(EIO base: procedures)+≡
  subroutine eio_test_init_in (eio, sample, process_ptr, data, success)
    class(eio_test_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    if (present (success)) success = .true.
  end subroutine eio_test_init_in

```

Switch from output to input. Again, nothing to do for the test type.

```

(EIO base: eio test: TBP)+≡
  procedure :: switch_inout => eio_test_switch_inout

(EIO base: procedures)+≡
  subroutine eio_test_switch_inout (eio, success)
    class(eio_test_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    if (present (success)) success = .true.
  end subroutine eio_test_switch_inout

```

Output. Increment the event counter and store the momenta of the current event.

```

(EIO base: eio test: TBP)+≡
  procedure :: output => eio_test_output

(EIO base: procedures)+≡
  subroutine eio_test_output (eio, event, i_prc)
    class(eio_test_t), intent(inout) :: eio
    type(event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    type(particle_set_t), pointer :: pset
    type(particle_t) :: prt
    eio%event_n = eio%event_n + 1
    eio%event_i = eio%event_n
    eio%i_prc = i_prc
    pset => event%get_particle_set_ptr ()
    prt = particle_set_get_particle (pset, 3)
    eio%event_p(1, eio%event_i) = particle_get_momentum (prt)
    prt = particle_set_get_particle (pset, 4)
    eio%event_p(2, eio%event_i) = particle_get_momentum (prt)
  end subroutine eio_test_output

```

Input. Increment the event counter and retrieve the momenta of the current event. For the test case, we do not actually modify the current event.

```

(EIO base: eio test: TBP)+≡
  procedure :: input_i_prc => eio_test_input_i_prc
  procedure :: input_event => eio_test_input_event

```

```

(EIO base: procedures)+≡
subroutine eio_test_input_i_prc (eio, i_prc, iostat)
  class(eio_test_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  i_prc = eio%i_prc
  iostat = 0
end subroutine eio_test_input_i_prc

subroutine eio_test_input_event (eio, event, iostat)
  class(eio_test_t), intent(inout) :: eio
  type(event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  type(particle_set_t), pointer :: pset
  type(particle_t) :: prt
  eio%event_i = eio%event_i + 1
  iostat = 0
end subroutine eio_test_input_event

```

## Test I/O methods

We test the implementation of all I/O methods using the `eio_test_t` type.

```

(EIO base: execute tests)≡
call test (eio_base_1, "eio_base_1", &
  "read and write event contents", &
  u, results)

(EIO base: tests)+≡
subroutine eio_base_1 (u)
  integer, intent(in) :: u
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  class(eio_t), allocatable :: eio
  integer :: i_prc, iostat
  type(string_t) :: sample

  write (u, "(A)")  "* Test output: eio_base_1"
  write (u, "(A)")  "* Purpose: generate and read/write an event"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  process_ptr%ptr => process
  allocate (process_instance)
  call prepare_test_process (process, process_instance)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%connect (process_instance)

```

```

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_test1"

allocate (eio_test_t :: eio)

call eio%init_out (sample, [process_ptr])
call event%generate (1, [0._default, 0._default])
call eio%output (event, 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])
call eio%input_i_prc (i_prc, iostat)
call eio%input_event (event, iostat)
call eio%write (u)
write (u, "(A)")
write (u, "(1x,A,I0)")  "i = ", i_prc

write (u, "(A)")
write (u, "(A)")  "* Generate and append another event"
write (u, "(A)")

call eio%switch_inout ()
call event%generate (1, [0._default, 0._default])
call eio%output (event, 5)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Re-read both events"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])
call eio%input_i_prc (i_prc, iostat)
call eio%input_event (event, iostat)
call eio%input_i_prc (i_prc, iostat)
call eio%input_event (event, iostat)
call eio%write (u)
write (u, "(A)")
write (u, "(1x,A,I0)")  "i = ", i_prc

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
deallocate (eio)

```



```

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_base_1"

end subroutine eio_base_1

```

## 17.3 Raw Event I/O

The raw format is for internal use only. All data are stored unformatted, so they can be efficiently be re-read on the same machine, but not necessarily on another machine.

```

<eio_raw.f90>≡
  <File header>

module eio_raw

  use kinds !NODEP!
  <Use file utils>
  <Use strings>
  use diagnostics !NODEP!
  use unit_tests

  use lorentz !NODEP!
  use particles
  use beams
  use processes
  use events
  use eio_data
  use eio_base

  <Standard module head>

  <EIO raw: public>

  <EIO raw: types>

contains

  <EIO raw: procedures>

  <EIO raw: tests>

end module eio_raw

```

### 17.3.1 Type

```
<EIO raw: public>≡
    public :: eio_raw_t

<EIO raw: types>≡
    type, extends (eio_t) :: eio_raw_t
        type(string_t) :: filename
        logical :: reading = .false.
        logical :: writing = .false.
        integer :: unit = 0
    contains
        <EIO raw: eio raw: TBP>
    end type eio_raw_t
```

Output. This is not the actual event format, but a readable account of the current object status.

```
<EIO raw: eio raw: TBP>≡
    procedure :: write => eio_raw_write

<EIO raw: procedures>≡
    subroutine eio_raw_write (object, unit)
        class(eio_raw_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "Raw event stream:"
        if (object%reading) then
            write (u, "(3x,A,A)") "Reading from file = ", char (object%filename)
        else if (object%writing) then
            write (u, "(3x,A,A)") "Writing to file   = ", char (object%filename)
        else
            write (u, "(3x,A)") "[closed]"
        end if
    end subroutine eio_raw_write
```

Finalizer: close any open file.

```
<EIO raw: eio raw: TBP>+≡
    procedure :: final => eio_raw_final

<EIO raw: procedures>+≡
    subroutine eio_raw_final (object)
        class(eio_raw_t), intent(inout) :: object
        if (object%reading .or. object%writing) then
            write (msg_buffer, "(A,A,A)") "Events: closing raw file '", &
                char (object%filename), "'"
            call msg_message ()
            close (object%unit)
            object%reading = .false.
            object%writing = .false.
        end if
    end subroutine eio_raw_final
```

Initialize event writing.

*(EIO raw: eio raw: TBP)*+≡

procedure :: init\_out => eio\_raw\_init\_out

*(EIO raw: procedures)*+≡

```
subroutine eio_raw_init_out (eio, sample, process_ptr, data, success)
  class(eio_raw_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  character(32) :: md5sum_prc, md5sum_cfg
  eio%filename = sample // ".evx"
  eio%unit = free_unit ()
  write (msg_buffer, "(A,A,A)") "Events: writing to raw file '", &
    char (eio%filename), ""
  call msg_message ()
  eio%writing = .true.
  if (present (data)) then
    md5sum_prc = data%md5sum_prc
    md5sum_cfg = data%md5sum_cfg
  else
    md5sum_prc = ""
    md5sum_cfg = ""
  end if
  open (eio%unit, file = char (eio%filename), form = "unformatted", &
    action = "write", status = "replace")
  write (eio%unit) md5sum_prc
  write (eio%unit) md5sum_cfg
  if (present (success)) success = .true.
end subroutine eio_raw_init_out
```

Initialize event reading.

*(EIO raw: eio raw: TBP)*+≡

procedure :: init\_in => eio\_raw\_init\_in

*(EIO raw: procedures)*+≡

```
subroutine eio_raw_init_in (eio, sample, process_ptr, data, success)
  class(eio_raw_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  character(32) :: md5sum_prc, md5sum_cfg
  eio%filename = sample // ".evx"
  eio%unit = free_unit ()
  write (msg_buffer, "(A,A,A)") "Events: reading from raw file '", &
    char (eio%filename), ""
  call msg_message ()
  eio%reading = .true.
  open (eio%unit, file = char (eio%filename), form = "unformatted", &
    action = "read", status = "old")
  read (eio%unit) md5sum_prc
  read (eio%unit) md5sum_cfg
```

```

if (present (success)) then
  success = .true.
  if (present (data)) then
    if (data%md5sum_prc /= "") then
      success = success .and. md5sum_prc == data%md5sum_prc
    end if
    if (data%md5sum_cfg /= "") then
      success = success .and. md5sum_cfg == data%md5sum_cfg
    end if
  end if
end if
end subroutine eio_raw_init_in

```

Switch from input to output: reopen the file for reading.

```

(EIO raw: eio raw: TBP)+≡
  procedure :: switch_inout => eio_raw_switch_inout

(EIO raw: procedures)+≡
  subroutine eio_raw_switch_inout (eio, success)
    class(eio_raw_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    write (msg_buffer, "(A,A,A)") "Events: appending to raw file '", &
      char (eio%filename), "'"
    call msg_message ()
    close (eio%unit, status = "keep")
    eio%reading = .false.
    open (eio%unit, file = char (eio%filename), form = "unformatted", &
      action = "write", position = "append", status = "old")
    eio%writing = .true.
    if (present (success)) success = .true.
  end subroutine eio_raw_switch_inout

```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

```

(EIO raw: eio raw: TBP)+≡
  procedure :: output => eio_raw_output

(EIO raw: procedures)+≡
  subroutine eio_raw_output (eio, event, i_prc)
    class(eio_raw_t), intent(inout) :: eio
    type(event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    type(particle_set_t), pointer :: pset
    if (eio%writing) then
      if (event%has_particle_set ()) then
        write (eio%unit) i_prc
        write (eio%unit) event%get_i_mci ()
        write (eio%unit) event%get_i_term ()
        write (eio%unit) event%get_channel ()
        write (eio%unit) event%get_weight ()
        write (eio%unit) event%get_sqme ()
        pset => event%get_particle_set_ptr ()
        call particle_set_write_raw (pset, eio%unit)
      else

```

```

        call msg_bug ("Event: write raw: particle set is undefined")
    end if
else
    call eio%write ()
    call msg_fatal ("Raw event file is not open for writing")
end if
end subroutine eio_raw_output

```

Input an event.

```

<EIO raw: eio raw: TBP>+=
    procedure :: input_i_prc => eio_raw_input_i_prc
    procedure :: input_event => eio_raw_input_event

<EIO raw: procedures>+=
    subroutine eio_raw_input_i_prc (eio, i_prc, iostat)
        class(eio_raw_t), intent(inout) :: eio
        integer, intent(out) :: i_prc
        integer, intent(out) :: iostat
        if (eio%reading) then
            read (eio%unit, iostat = iostat) i_prc
        else
            call eio%write ()
            call msg_fatal ("Raw event file is not open for reading")
        end if
    end subroutine eio_raw_input_i_prc

    subroutine eio_raw_input_event (eio, event, iostat)
        class(eio_raw_t), intent(inout) :: eio
        type(event_t), intent(inout), target :: event
        integer, intent(out) :: iostat
        type(particle_set_t), pointer :: pset
        integer :: i_mci, i_term, channel
        real(default) :: weight, sqme
        if (eio%reading) then
            read (eio%unit, iostat = iostat) i_mci
            if (iostat /= 0) return
            read (eio%unit, iostat = iostat) i_term
            if (iostat /= 0) return
            read (eio%unit, iostat = iostat) channel
            if (iostat /= 0) return
            read (eio%unit, iostat = iostat) weight
            if (iostat /= 0) return
            read (eio%unit, iostat = iostat) sqme
            if (iostat /= 0) return
            call event%reset ()
            call event%select (i_mci, i_term, channel)
            call event%set_sqme (sqme, weight)
            pset => event%get_particle_set_ptr ()
            call particle_set_read_raw (pset, eio%unit, iostat)
            if (iostat /= 0) return
            if (associated (event%process)) then
                call particle_set_set_model (pset, event%process%get_model_ptr ())
            end if
            call event%accept_particle_set ()
        end if
    end subroutine eio_raw_input_event

```

```

else
    call eio%write ()
    call msg_fatal ("Raw event file is not open for reading")
end if
end subroutine eio_raw_input_event

```

### 17.3.2 Unit tests

```

<EIO raw: public>+≡
    public :: eio_raw_test

<EIO raw: tests>≡
    subroutine eio_raw_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <EIO raw: execute tests>
    end subroutine eio_raw_test

```

#### Test I/O methods

We test the implementation of all I/O methods.

```

<EIO raw: execute tests>≡
    call test (eio_raw_1, "eio_raw_1", &
        "read and write event contents", &
        u, results)

<EIO raw: tests>+≡
    subroutine eio_raw_1 (u)
        integer, intent(in) :: u
        type(event_t), allocatable, target :: event
        type(process_t), allocatable, target :: process
        type(process_ptr_t) :: process_ptr
        type(process_instance_t), allocatable, target :: process_instance
        class(eio_t), allocatable :: eio
        integer :: i_prc, iostat
        type(string_t) :: sample

        write (u, "(A)")  "* Test output: eio_raw_1"
        write (u, "(A)")  "* Purpose: generate and read/write an event"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize test process"

        allocate (process)
        process_ptr%ptr => process
        allocate (process_instance)
        call prepare_test_process (process, process_instance)
        call process_instance%setup_event_data ()

        allocate (event)
        call event%connect (process_instance)

```

```

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_raw_1"

allocate (eio_raw_t :: eio)

call eio%init_out (sample, [process_ptr])
call event%generate (1, [0._default, 0._default])
call event%write (u)
write (u, "(A)")

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%connect (process_instance)

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0) write (u, "(A,I0)")  "I/O error (i_prc):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0) write (u, "(A,I0)")  "I/O error (event):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)")  "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate and append another event"
write (u, "(A)")

call eio%switch_inout ()
call event%generate (1, [0._default, 0._default])
call event%write (u)
write (u, "(A)")

```

```

call eio%output (event, i_prc = 5)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read both events"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%connect (process_instance)

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0) write (u, "(A,I0)") "I/O error (i_prc/1):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0) write (u, "(A,I0)") "I/O error (event/1):", iostat
call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0) write (u, "(A,I0)") "I/O error (i_prc/2):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0) write (u, "(A,I0)") "I/O error (event/2):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)") "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
deallocate (eio)

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_raw_1"

end subroutine eio_raw_1

```



## 17.4 HEP Common Blocks

Long ago, to transfer data between programs one had to set up a common block and link both programs as libraries to the main executable. The HEP community standardizes several of those common blocks.

The modern way of data exchange uses data files with standard formats. However, the LHEF standard data format derives from a common block (actually, two).

WHIZARD used to support those common blocks, and LHEF was implemented via writing/reading blocks. We still keep this convention, but intend to eliminate common blocks (or any other static storage) from the workflow in the future. This will gain flexibility towards concurrent running of program images.

We encapsulate everything here in a module. The module holds the variables which are part of the common block.

Note: This code is taken essentially unchanged from WHIZARD 2.1 and does not (yet) provide unit tests.

```
(hep_common.f90)≡  
  ⟨File header⟩  
  
  module hep_common  
  
    use kinds !NODEP!  
    ⟨Use file utils⟩  
    ⟨Use strings⟩  
    use diagnostics !NODEP!  
  
    use lorentz !NODEP!  
    use polarizations  
    use particles  
    use subevents  
    use events  
  
    ⟨Standard module head⟩  
  
    ⟨HEP common: public⟩  
  
    ⟨HEP common: interfaces⟩  
  
    ⟨HEP common: parameters⟩  
  
    ⟨HEP common: variables⟩  
  
    ⟨HEP common: common blocks⟩  
  
    contains  
  
    ⟨HEP common: procedures⟩  
  
  end module hep_common
```

### 17.4.1 Event characteristics

The maximal number of particles in an event record.

$\langle \text{HEP common: parameters} \rangle \equiv$   
integer, parameter :: MAXNUP = 500

The number of particles in this event.

$\langle \text{HEP common: variables} \rangle \equiv$   
integer :: NUP

The process ID for this event.

$\langle \text{HEP common: variables} \rangle + \equiv$   
integer :: IDPRUP

The weight of this event ( $\pm 1$  for unweighted events).

$\langle \text{HEP common: variables} \rangle + \equiv$   
double precision :: XWGTUP

The factorization scale that is used for PDF calculation ( $-1$  if undefined).

$\langle \text{HEP common: variables} \rangle + \equiv$   
double precision :: SCALUP

The QED and QCD couplings  $\alpha$  used for this event ( $-1$  if undefined).

$\langle \text{HEP common: variables} \rangle + \equiv$   
double precision :: AQEDUP  
double precision :: AQCDUP

### 17.4.2 Particle characteristics

The PDG code:

$\langle \text{HEP common: variables} \rangle + \equiv$   
integer, dimension(MAXNUP) :: IDUP

The status code. Incoming:  $-1$ , outgoing:  $+1$ . Intermediate t-channel propagator:  $-2$  (currently not used by WHIZARD). Intermediate resonance whose mass should be preserved:  $2$ . Intermediate resonance for documentation:  $3$  (currently not used). Beam particles:  $-9$ .

$\langle \text{HEP common: variables} \rangle + \equiv$   
integer, dimension(MAXNUP) :: ISTUP

Index of first and last mother.

$\langle \text{HEP common: variables} \rangle + \equiv$   
integer, dimension(2,MAXNUP) :: MOTHUP

Color line index of the color and anticolor entry for the particle. The standard recommends using large numbers; we start from MAXNUP+1.

$\langle \text{HEP common: variables} \rangle + \equiv$   
integer, dimension(2,MAXNUP) :: ICOLUP

Momentum, energy, and invariant mass:  $(p_x, p_y, p_z, E, M)$ . For space-like particles,  $M$  is the negative square root of the absolute value of the invariant mass.

$\langle \text{HEP common: variables} \rangle + \equiv$   
double precision, dimension(5,MAXNUP) :: PUP

Invariant lifetime (distance) from production to decay in mm.

$\langle \text{HEP common: variables} \rangle + \equiv$   
double precision, dimension(MAXNUP) :: VTIMUP

Cosine of the angle between the spin-vector and a particle and the 3-momentum of its mother, given in the lab frame. If undefined/unpolarized: 9.

```
<HEP common: variables>+≡
    double precision, dimension(MAXNUP) :: SPINUP
```

### 17.4.3 The HEPUP common block

This common block is filled once per run.

#### Run characteristics

The maximal number of different processes.

```
<HEP common: parameters>+≡
    integer, parameter :: MAXPUP = 100
```

The beam PDG codes.

```
<HEP common: variables>+≡
    integer, dimension(2) :: IDBMUP
```

The beam energies in GeV.

```
<HEP common: variables>+≡
    double precision, dimension(2) :: EBMUP
```

The PDF group and set for the two beams. (Undefined: use  $-1$ ; LHAPDF: use group = 0).

```
<HEP common: variables>+≡
    integer, dimension(2) :: PDFGUP
    integer, dimension(2) :: PDFSUP
```

The (re)weighting model. 1: events are weighted, the shower generator (SHG) selects processes according to the maximum weight (in pb) and unweights events. 2: events are weighted, the SHG selects processes according to their cross section (in pb) and unweights events. 3: events are unweighted and simply run through the SHG. 4: events are weighted, and the SHG keeps the weight. Negative numbers: negative weights are allowed (and are reweighted to  $\pm 1$  by the SHG, if allowed).

WHIZARD only supports modes 3 and 4, as the SHG is not given control over process selection. This is consistent with writing events to file, for offline showering.

```
<HEP common: variables>+≡
    integer :: IDWTUP
```

The number of different processes.

```
<HEP common: variables>+≡
    integer :: NPRUP
```

#### Process characteristics

Cross section and error in pb. (Cross section is needed only for IDWTUP = 2, so here both values are given for informational purposes only.)

```
<HEP common: variables>+≡
    double precision, dimension(MAXPUP) :: XSECUP
    double precision, dimension(MAXPUP) :: XERRUP
```

Maximum weight, i.e., the maximum value that XWGTUP can take. Also unused for the supported weighting models. It is  $\pm 1$  for unweighted events.

```
<HEP common: variables>+≡
    double precision, dimension(MAXPUP) :: XMAXUP
```

Internal ID of the selected process, matches IDPRUP below.

```
<HEP common: variables>+≡
    integer, dimension(MAXPUP) :: LPRUP
```

### The common block

```
<HEP common: common blocks>≡
    common /HEPRUP/ &
        IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP, &
        XSECUP, XERRUP, XMAXUP, LPRUP
    save /HEPRUP/
```

Fill the run characteristics of the common block. The initialization sets the beam properties, number of processes, and weighting model.

```
<HEP common: public>≡
    public :: heprup_init

<HEP common: procedures>≡
    subroutine heprup_init &
        (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
        integer, dimension(2), intent(in) :: beam_pdg
        real(default), dimension(2), intent(in) :: beam_energy
        integer, intent(in) :: n_processes
        logical, intent(in) :: unweighted
        logical, intent(in) :: negative_weights
        IDBMUP = beam_pdg
        EBMUP = beam_energy
        PDFGUP = -1
        PDFSUP = -1
        if (unweighted) then
            IDWTUP = 3
        else
            IDWTUP = 4
        end if
        if (negative_weights) IDWTUP = - IDWTUP
        NPRUP = n_processes
    end subroutine heprup_init
```

Specify PDF set info. Since we support only LHAPDF, the group entry is zero.

```
<HEP common: public>+≡
    public :: heprup_set_lhapdf_id

<HEP common: procedures>+≡
    subroutine heprup_set_lhapdf_id (i_beam, pdf_id)
        integer, intent(in) :: i_beam, pdf_id
        PDFGUP(i_beam) = 0
        PDFSUP(i_beam) = pdf_id
    end subroutine heprup_set_lhapdf_id
```

Fill the characteristics for a particular process. Only the process ID is mandatory. Note that WHIZARD computes cross sections in fb, so we have to rescale to pb. The maximum weight is meaningless for unweighted events.

```

<HEP common: public>+≡
  public :: heprup_set_process_parameters

<HEP common: procedures>+≡
  subroutine heprup_set_process_parameters &
    (i, process_id, cross_section, error, max_weight)
    integer, intent(in) :: i, process_id
    real(default), intent(in), optional :: cross_section, error, max_weight
    real(default), parameter :: pb_per_fb = 1.e-3_default
    LPRUP(i) = process_id
    if (present (cross_section)) then
      XSECUP(i) = cross_section * pb_per_fb
    else
      XSECUP(i) = 0
    end if
    if (present (error)) then
      XERRUP(i) = error * pb_per_fb
    else
      XERRUP(i) = 0
    end if
    select case (IDWTUP)
    case (3); XMAXUP(i) = 1
    case (4)
      if (present (max_weight)) then
        XMAXUP(i) = max_weight * pb_per_fb
      else
        XMAXUP(i) = 0
      end if
    end select
  end subroutine heprup_set_process_parameters

```

#### 17.4.4 Run parameter output (verbose)

This is a verbose output of the HEPRUP block.

```

<HEP common: public>+≡
  public :: heprup_write_verbose

<HEP common: procedures>+≡
  subroutine heprup_write_verbose (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    write (u, "(A)") "HEPRUP Common Block"
    write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "IDBMUP", IDBMUP, &
      "PDG code of beams"
    write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)") "EBMUP ", EBMUP, &
      "Energy of beams in GeV"
    write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "PDFGUP", PDFGUP, &
      "PDF author group [-1 = undefined]"
    write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "PDFSUP", PDFSUP, &

```

```

      "PDF set ID          [-1 = undefined]"
write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)") "IDWTUP", IDWTUP, &
      "LHA code for event weight mode"
write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)") "NPRUP ", NPRUP, &
      "Number of user subprocesses"
do i = 1, NPRUP
  write (u, "(1x,A,I0)") "Subprocess #", i
  write (u, "(3x,A6,' = ',F12.5,1x,12x,8x,A)") "XSECUP", XSECUP(i), &
    "Cross section in pb"
  write (u, "(3x,A6,' = ',F12.5,1x,12x,8x,A)") "XERRUP", XERRUP(i), &
    "Cross section error in pb"
  write (u, "(3x,A6,' = ',F12.5,1x,12x,8x,A)") "XMAXUP", XMAXUP(i), &
    "Maximum event weight (cf. IDWTUP)"
  write (u, "(3x,A6,' = ',I9,3x,1x,12x,8x,A)") "LPRUP ", LPRUP(i), &
    "Subprocess ID"
end do
end subroutine heprup_write_verbose

```

#### 17.4.5 Run parameter output (other formats)

This routine writes the initialization block according to the LHEF standard. It uses the current contents of the HEPRUP block.

```

<HEP common: public>+≡
  public :: heprup_write_lhef

<HEP common: procedures>+≡
  subroutine heprup_write_lhef (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    write (u, '(A)') "<init>"
    write (u, "(2(1x,I0),2(1x,ES17.10),6(1x,I0))") &
      IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
    do i = 1, NPRUP
      write (u, "(3(1x,ES17.10),1x,I0)") &
        XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
    end do
    write (u, '(A)') "</init>"
  end subroutine heprup_write_lhef

```

#### 17.4.6 The HEPEUP common block

```

<HEP common: common blocks>+≡
  common /HEPEUP/ &
    NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP, &
    IDUP, ISTUP, MOTHUP, ICOLUP, PUP, VTIMUP, SPINUP
  save /HEPEUP/

```

Fill the event characteristics of the common block. The initialization sets only the number of particles and initializes the rest with default values. The other routine sets the optional parameters.

```

<HEP common: public>+≡
  public :: hepeup_init
  public :: hepeup_set_event_parameters

<HEP common: procedures>+≡
  subroutine hepeup_init (n_tot)
    integer, intent(in) :: n_tot
    NUP = n_tot
    IDPRUP = 0
    XWGTUP = 1
    SCALUP = -1
    AQEDUP = -1
    AQCDUP = -1
  end subroutine hepeup_init

  subroutine hepeup_set_event_parameters &
    (proc_id, weight, scale, alpha_qed, alpha_qcd)
    integer, intent(in), optional :: proc_id
    real(default), intent(in), optional :: weight, scale, alpha_qed, alpha_qcd
    if (present (proc_id)) IDPRUP = proc_id
    if (present (weight)) XWGTUP = weight
    if (present (scale)) SCALUP = scale
    if (present (alpha_qed)) AQEDUP = alpha_qed
    if (present (alpha_qcd)) AQCDUP = alpha_qcd
  end subroutine hepeup_set_event_parameters

```

Below we need the particle status codes which are actually defined in the `subevents` module.

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

```

<HEP common: public>+≡
  public :: hepeup_set_particle

<HEP common: procedures>+≡
  subroutine hepeup_set_particle (i, pdg, status, parent, col, p, m2)
    integer, intent(in) :: i
    integer, intent(in) :: pdg, status
    integer, dimension(:), intent(in) :: parent
    type(vector4_t), intent(in) :: p
    integer, dimension(2), intent(in) :: col
    real(default), intent(in) :: m2
    if (i > MAXNUP) then
      call msg_error (arr=(/ &
var_str ("Too many particles in HEPEUP common block. If this happened "), &
var_str ("during event output, your events will be invalid; please consider "), &
var_str ("switching to a modern event format like HEPMC. If you are not "), &
var_str ("using an old, HEPEUP based format and nevertheless get this error,"), &
var_str ("please notify the WHIZARD developers,") &
/))
      return
    end if

```

```

IDUP(i) = pdg
select case (status)
case (PRT_BEAM);          ISTUP(i) = -9
case (PRT_INCOMING);      ISTUP(i) = -1
case (PRT_BEAM_REMNANT);  ISTUP(i) = 3
case (PRT_OUTGOING);      ISTUP(i) = 1
case (PRT_RESONANT);      ISTUP(i) = 2
case (PRT_VIRTUAL);       ISTUP(i) = 3
case default;             ISTUP(i) = 0
end select
select case (size (parent))
case (0);      MOTHUP(:,i) = 0
case (1);      MOTHUP(1,i) = parent(1); MOTHUP(2,i) = 0
case default;  MOTHUP(:,i) = (/ parent(1), parent(size (parent)) /)
end select
if (col(1) > 0) then
    ICOLUP(1,i) = 500 + col(1)
else
    ICOLUP(1,i) = 0
end if
if (col(2) > 0) then
    ICOLUP(2,i) = 500 + col(2)
else
    ICOLUP(2,i) = 0
end if
PUP(1:3,i) = vector3_get_components (space_part (p))
PUP(4,i) = energy (p)
PUP(5,i) = sign (sqrt (abs (m2)), m2)
VTIMUP(i) = 0
SPINUP(i) = 9
end subroutine hepeup_set_particle

```

Set the lifetime, actually  $c\tau$  measured in mm, where  $\tau$  is the invariant lifetime.

```

<HEP common: public>+≡
    public :: hepeup_set_particle_lifetime

<HEP common: procedures>+≡
    subroutine hepeup_set_particle_lifetime (i, lifetime)
        integer, intent(in) :: i
        real(default), intent(in) :: lifetime
        VTIMUP(i) = lifetime
    end subroutine hepeup_set_particle_lifetime

```

Set the particle spin entry. We need the cosine of the angle of the spin axis with respect to the three-momentum of the parent particle.

If the particle has a full polarization density matrix given, we need the particle momentum and polarization as well as the mother-particle momentum. The polarization is transformed into a spin vector (which is sensible only for spin-1/2 or massless particles), which then is transformed into the lab frame (by a rotation of the 3-axis to the particle momentum axis). Finally, we compute the scalar product of this vector with the mother-particle three-momentum.

This puts severe restrictions on the applicability of this definition, and Lorentz invariance is lost. Unfortunately, the Les Houches Accord requires this



computation.

```

<HEP common: public>+≡
    public :: hepeup_set_particle_spin

<HEP common: interfaces>≡
    interface hepeup_set_particle_spin
        module procedure hepeup_set_particle_spin_pol
    end interface

<HEP common: procedures>+≡
    subroutine hepeup_set_particle_spin_pol (i, p, pol, p_mother)
        integer, intent(in) :: i
        type(vector4_t), intent(in) :: p
        type(polarization_t), intent(in) :: pol
        type(vector4_t), intent(in) :: p_mother
        type(vector3_t) :: s3, p3
        type(vector4_t) :: s4
        s3 = vector3_moving (polarization_get_axis (pol))
        p3 = space_part (p)
        s4 = rotation_to_2nd (3, p3) * vector4_moving (0._default, s3)
        SPINUP(i) = enclosed_angle_ct (s4, p_mother)
    end subroutine hepeup_set_particle_spin_pol

```

#### 17.4.7 The HEPEVT common block

For the LEP Monte Carlos, a standard common block has been proposed in AKV89. We strongly recommend its use. (The description is an abbreviated transcription of AKV89, Vol. 3, pp. 327-330).

NMXHEP is the maximum number of entries:

```

<HEP common: variables>+≡
    integer, parameter :: NMXHEP = 4000

```

NEVHEP is normally the event number, but may take special values as follows:

0 the program does not keep track of event numbers. -1 a special initialization record. -2 a special final record.

```

<HEP common: variables>+≡
    integer :: NEVHEP

```

NHEP holds the number of entries for this event.

```

<HEP common: variables>+≡
    integer :: NHEP

```

The entry ISTHEP(N) gives the status code for the Nth entry, with the following semantics: 0 a null entry. 1 an existing entry, which has not decayed or fragmented. 2 a decayed or fragmented entry, which is retained for event history information. 3 documentation line. 4- 10 reserved for future standards. 11-200 at the disposal of each model builder. 201- at the disposal of users.

```

<HEP common: variables>+≡
    integer, dimension(NMXHEP) :: ISTHEP

```

The Particle Data Group has proposed standard particle codes, which are to be stored in IDHEP(N).

```
<HEP common: variables>+≡
integer, dimension(NMXHEP) :: IDHEP
```

JMOHEP(1,N) points to the mother of the Nth entry, if any. It is set to zero for initial entries. JMOHEP(2,N) points to the second mother, if any.

```
<HEP common: variables>+≡
integer, dimension(2, NMXHEP) :: JMOHEP
```

JDAHEP(1,N) and JDAHEP(2,N) point to the first and last daughter of the Nth entry, if any. These are zero for entries which have not yet decayed. The other daughters are stored in between these two.

```
<HEP common: variables>+≡
integer, dimension(2, NMXHEP) :: JDAHEP
```

In PHEP we store the momentum of the particle, more specifically this means that PHEP(1,N), PHEP(2,N), and PHEP(3,N) contain the momentum in the  $x$ ,  $y$ , and  $z$  direction (as defined by the machine people), measured in GeV/c. PHEP(4,N) contains the energy in GeV and PHEP(5,N) the mass in GeV/ $c^2$ . The latter may be negative for spacelike partons.

```
<HEP common: variables>+≡
double precision, dimension(5, NMXHEP) :: PHEP
```

Finally VHEP is the place to store the position of the production vertex. VHEP(1,N), VHEP(2,N), and VHEP(3,N) contain the  $x$ ,  $y$ , and  $z$  coordinate (as defined by the machine people), measured in mm. VHEP(4,N) contains the production time in mm/c.

```
<HEP common: variables>+≡
double precision, dimension(4, NMXHEP) :: VHEP
```

As an amendment to the proposed standard common block HEPEVT, we also have a polarisation common block HEPSPN, as described in AKV89. SHEP(1,N), SHEP(2,N), and SHEP(3,N) give the  $x$ ,  $y$ , and  $z$  component of the spinvector  $s$  of a fermion in the fermions restframe.

Furthermore, we add the polarization of the corresponding outgoing particles:

```
<HEP common: variables>+≡
integer, dimension(NMXHEP) :: hepevt_pol
```

By convention, SHEP(4,N) is always 1. All this is taken from StdHep 4.06 manual and written using Fortran90 conventions.

```
<HEP common: common blocks>+≡
common /HEPEVT/ &
NEVHEP, NHEP, ISTHEP, IDHEP, &
JMOHEP, JDAHEP, PHEP, VHEP
save /HEPEVT/
```

Here we store HEPEVT parameters of the WHIZARD 1 realization which are not part of the HEPEVT common block.

```

<HEP common: variables>+≡
    integer :: hepevt_n_out, hepevt_n_remnants

<HEP common: variables>+≡
    double precision :: hepevt_weight, hepevt_function_value
    double precision :: hepevt_function_ratio

```

Filling HEPEVT: If the event count is not provided, set NEVHEP to zero. If the event count is -1 or -2, the record corresponds to initialization and finalization, and the event is irrelevant.

Note that the event count may be larger than  $2^{31}$  (2 GEvents). In that case, cut off the upper bits since NEVHEP is probably limited to default integer.

```

<HEP common: public>+≡
    public :: hepevt_init
    public :: hepevt_set_event_parameters

<HEP common: procedures>+≡
    subroutine hepevt_init (n_tot, n_out)
        integer, intent(in) :: n_tot, n_out
        NHEP                = n_tot
        NEVHEP              = 0
        hepevt_n_out        = n_out
        hepevt_n_remnants = 0
        hepevt_weight       = 1
        hepevt_function_value = 0
        hepevt_function_ratio = 1
    end subroutine hepevt_init

    subroutine hepevt_set_event_parameters &
        (weight, function_value, function_ratio, i_evt)
        integer, intent(in), optional :: i_evt
        real(default), intent(in), optional :: weight, function_value, &
            function_ratio
        integer(i32), parameter :: huge32 = huge (0_i32)
        if (present (i_evt)) NEVHEP = i_evt
        if (present (weight)) hepevt_weight = weight
        if (present (function_value)) hepevt_function_value = &
            function_value
        if (present (function_ratio)) hepevt_function_ratio = &
            function_ratio
    end subroutine hepevt_set_event_parameters

```

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

```

<HEP common: public>+≡
    public :: hepevt_set_particle

<HEP common: procedures>+≡
    subroutine hepevt_set_particle (i, pdg, status, parent, child, p, m2, hel)
        integer, intent(in) :: i
        integer, intent(in) :: pdg, status

```

```

integer, dimension(:), intent(in) :: parent
integer, dimension(:), intent(in) :: child
type(vector4_t), intent(in) :: p
real(default), intent(in) :: m2
integer, intent(in) :: hel
IDHEP(i) = pdg
select case (status)
  case (PRT_BEAM);      ISTHEP(i) = 2
  case (PRT_INCOMING);  ISTHEP(i) = 2
  case (PRT_OUTGOING);  ISTHEP(i) = 1
  case (PRT_VIRTUAL);   ISTHEP(i) = 2
  case (PRT_RESONANT);  ISTHEP(i) = 2
  case default;         ISTHEP(i) = 0
end select
select case (size (parent))
case (0);      JMOHEP(:,i) = 0
case (1);      JMOHEP(1,i) = parent(1); JMOHEP(2,i) = 0
case default;  JMOHEP(:,i) = (/ parent(1), parent(size (parent)) /)
end select
select case (size (child))
case (0);      JDAHEP(:,i) = 0
case (1);      JDAHEP(:,i) = child(1)
case default;  JDAHEP(:,i) = (/ child(1), child(size (child)) /)
end select
PHEP(1:3,i) = vector3_get_components (space_part (p))
PHEP(4,i) = energy (p)
PHEP(5,i) = sign (sqrt (abs (m2)), m2)
VHEP(1:4,i) = 0
hepevt_pol(i) = hel
end subroutine hepevt_set_particle

```

## 17.4.8 Event output

This is a verbose output of the HEPEVT block.

```

<HEP common: public>+≡
  public :: hepevt_write_verbose

<HEP common: procedures>+≡
  subroutine hepevt_write_verbose (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    write (u, "(A)") "HEPEVT Common Block"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "NEVHEP", NEVHEP, &
      "Event number"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "NHEP ", NHEP, &
      "Number of particles in event"
    do i = 1, NHEP
      write (u, "(1x,A,I0)") "Particle #", i
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
        "ISTHEP", ISTHEP(i), "Status code: "
      select case (ISTHEP(i))
      case ( 0); write (u, "(A)") "null entry"

```

```

case ( 1); write (u, "(A)") "outgoing"
case ( 2); write (u, "(A)") "decayed"
case ( 3); write (u, "(A)") "documentation"
case (4:10); write (u, "(A)") "[unspecified]"
case (11:200); write (u, "(A)") "[model-specific]"
case (201:); write (u, "(A)") "[user-defined]"
case default; write (u, "(A)") "[undefined]"
end select
write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "IDHEP ", IDHEP(i), &
"PDG code of particle"
write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "JMOHEP", JMOHEP(:,i), &
"Index of first/second mother"
write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "JDAHEP", JDAHEP(:,i), &
"Index of first/last daughter"
write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)") "PHEP12", PHEP(1:2,i), &
"Transversal momentum (x/y) in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PHEP3 ", PHEP(3,i), &
"Longitudinal momentum (z) in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PHEP4 ", PHEP(4,i), &
"Energy in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PHEP5 ", PHEP(5,i), &
"Invariant mass in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)") "VHEP12", VHEP(1:2,i), &
"Transversal displacement (xy) in mm"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "VHEP3 ", VHEP(3,i), &
"Longitudinal displacement (z) in mm"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "VHEP4 ", VHEP(4,i), &
"Production time in mm"
end do
end subroutine hepeup_write_verbose

```

This is a verbose output of the HEPEUP block.

```

<HEP common: public>+≡
public :: hepeup_write_verbose

<HEP common: procedures>+≡
subroutine hepeup_write_verbose (unit)
integer, intent(in), optional :: unit
integer :: u, i
u = output_unit (unit); if (u < 0) return
write (u, "(A)") "HEPEUP Common Block"
write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "NUP ", NUP, &
"Number of particles in event"
write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "IDPRUP", IDPRUP, &
"Subprocess ID"
write (u, "(3x,A6,' = ',G12.5,1x,20x,A)") "XWGTUP", XWGTUP, &
"Event weight"
write (u, "(3x,A6,' = ',G12.5,1x,20x,A)") "SCALUP", SCALUP, &
"Event energy scale in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,20x,A)") "AQEDUP", AQEDUP, &
"QED coupling [-1 = undefined]"
write (u, "(3x,A6,' = ',G12.5,1x,20x,A)") "AQCDUP", AQCDUP, &
"QCD coupling [-1 = undefined]"
do i = 1, NUP

```

```

write (u, "(1x,A,I0)") "Particle #", i
write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)") "IDUP ", IDUP(i), &
    "PDG code of particle"
write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
    "ISTUP ", ISTUP(i), "Status code: "
select case (ISTUP(i))
case (-1); write (u, "(A)") "incoming"
case ( 1); write (u, "(A)") "outgoing"
case (-2); write (u, "(A)") "spacelike"
case ( 2); write (u, "(A)") "resonance"
case ( 3); write (u, "(A)") "resonance (doc)"
case (-9); write (u, "(A)") "beam"
case default; write (u, "(A)") "[undefined]"
end select
write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "MOTHUP", MOTHUP(:,i), &
    "Index of first/last mother"
write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)") "ICOLUP", ICOLUP(:,i), &
    "Color/anticolor flow index"
write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)") "PUP1/2", PUP(1:2,i), &
    "Transversal momentum (x/y) in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PUP3 ", PUP(3,i), &
    "Longitudinal momentum (z) in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PUP4 ", PUP(4,i), &
    "Energy in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "PUP5 ", PUP(5,i), &
    "Invariant mass in GeV"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "VTIMUP", VTIMUP(i), &
    "Invariant lifetime in mm"
write (u, "(3x,A6,' = ',G12.5,1x,12x,8x,A)") "SPINUP", SPINUP(i), &
    "cos(spin angle) [9 = undefined]"
end do
end subroutine hepeup_write_verbose

```

#### 17.4.9 Event output in various formats

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

```

<HEP common: public>+=
public :: hepeup_write_lhef
public :: hepeup_write_lha

<HEP common: procedures>+=
subroutine hepeup_write_lhef (unit)
integer, intent(in), optional :: unit
integer :: u, i
u = output_unit (unit); if (u < 0) return
write (u, '(A)') "<event>"
write (u, "(2(1x,I0),4(1x,ES17.10))") &
    NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
do i = 1, NUP
write (u, "(6(1x,I0),7(1x,ES17.10))") &
    IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
    PUP(:,i), VTIMUP(i), SPINUP(i)

```

```

        end do
        write (u, '(A)') "</event>"
    end subroutine hepeup_write_lhef

    subroutine hepeup_write_lha (unit)
        integer, intent(in), optional :: unit
        integer :: u, i
        integer, dimension(MAXNUP) :: spin_up
        spin_up = SPINUP
        u = output_unit (unit); if (u < 0) return
16 format(2(1x,I5),1x,F17.10,3(1x,F13.6))
17 format(500(1x,I5))
18 format(1x,I5,4(1x,F17.10))
        write (u, 16) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
        write (u, 17) IDUP(:NUP)
        write (u, 17) MOTHUP(1,:NUP)
        write (u, 17) MOTHUP(2,:NUP)
        write (u, 17) ICOLUP(1,:NUP)
        write (u, 17) ICOLUP(2,:NUP)
        write (u, 17) ISTUP(:NUP)
        write (u, 17) spin_up(:NUP)
        do i = 1, NUP
            write (u, 18) i, PUP((/ 4,1,2,3 /), i)
        end do

    end subroutine hepeup_write_lha

```

This routine writes event output according to the HEPEVT standard. It uses the current contents of the HEPEVT block and some additional parameters according to the standard in WHIZARD 1. For the long ASCII format, the value of the sample function (i.e. the product of squared matrix element, structure functions and phase space factor is printed out). The option of reweighting matrix elements with respect to some reference cross section is not implemented in WHIZARD 2, therefore the second entry in the long ASCII format (the function ratio) is always one. The ATHENA format is an implementation of the HEPEVT format that is readable by the ATLAS ATHENA software framework. It is very similar to the WHIZARD 1 HEPEVT format, except that it contains an event counter, a particle counter inside the event, and has the HEPEVT ISTHEP status before the PDG code. The MOKKA format is a special ASCII format that contains the information to be parsed to the MOKKA LC fast simulation software.

```

<HEP common: public>+≡
    public :: hepevt_write_hepevt
    public :: hepevt_write_ascii
    public :: hepevt_write_athena
    public :: hepevt_write_mokka

<HEP common: procedures>+≡
    subroutine hepevt_write_hepevt (unit)
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit); if (u < 0) return
        write (u, *) NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight

```

```

do i = 1, NHEP
  write (u, *) ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i), &
    hepevt_pol(i)
  write (u, *) PHEP(:,i)
  write (u, *) VHEP(:,i), 0.d0
end do
end subroutine hepevt_write_hepevt

subroutine hepevt_write_ascii (unit, long)
  integer, intent(in), optional :: unit
  logical, intent(in) :: long
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, *) NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
    write (u, *) IDHEP(i), hepevt_pol(i)
    write (u, *) PHEP(:,i)
  end do
  if (long) write (u, *) hepevt_function_value, hepevt_function_ratio
end subroutine hepevt_write_ascii

subroutine hepevt_write_athena (unit, i_evt)
  integer, intent(in), optional :: unit, i_evt
  integer :: u, i, num_event
  num_event = 0
  if (present (i_evt)) num_event = i_evt
  u = output_unit (unit); if (u < 0) return
  write (u, *) num_event, NHEP
  do i = 1, NHEP
    write (u, *) i, ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i)
    write (u, *) PHEP(:,i)
    write (u, *) VHEP(1:4,i)
  end do
end subroutine hepevt_write_athena

subroutine hepevt_write_mokka (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  write (u, *) NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
    write (u, *) ISTHEP(i), IDHEP(i), JDAHEP(1,i), JDAHEP(2,i), &
      PHEP(1:3,i), PHEP(5,i)
  end do
end subroutine hepevt_write_mokka

```

## 17.4.10 Data Transfer: particle sets

The WHIZARD format for handling particle data in events is `particle_set_t`. We have to interface this to the common blocks.

We first create a new particle set that contains only the particles that are supported by the LHEF format. These are: beam, incoming, resonant, outgoing.



We drop particles with unknown, virtual or beam-remnant status.

From this set we fill the common block. Event information such as process ID and weight is not transferred here; this has to be done by the caller. The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

```

<HEP common: public>+≡
  public :: hepeup_from_particle_set

<HEP common: procedures>+≡
  subroutine hepeup_from_particle_set (particle_set, keep_beams)
    type(particle_set_t), intent(in), target :: particle_set
    logical, intent(in), optional :: keep_beams
    type(particle_t), pointer :: prt
    type(particle_set_t), target :: pset_hepevt
    integer :: i, n_parents
    integer, dimension(1) :: i_mother
    call particle_set_to_hepevt_form (particle_set, pset_hepevt, keep_beams)
    call hepeup_init (pset_hepevt%n_tot)
    do i = 1, pset_hepevt%n_tot
      prt => pset_hepevt%prt(i)
      call hepeup_set_particle (i, &
        particle_get_pdg (prt), &
        particle_get_status (prt), &
        particle_get_parents (prt), &
        particle_get_color (prt), &
        particle_get_momentum (prt), &
        particle_get_p2 (prt))
      n_parents = particle_get_n_parents (prt)
      if (n_parents == 1) then
        i_mother = particle_get_parents (prt)
        select case (particle_get_polarization_status (prt))
          case (PRT_GENERIC_POLARIZATION)
            call hepeup_set_particle_spin (i, &
              particle_get_momentum (prt), &
              particle_get_polarization (prt), &
              particle_get_momentum (pset_hepevt%prt(i_mother(1))))
        end select
      end if
    end do
    call particle_set_final (pset_hepevt)
  end subroutine hepeup_from_particle_set

```

The HEPEVT common block is quite similar, but does contain less information, e.g. no color flows (it was LEP time). The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

```

<HEP common: public>+≡
  public :: hepevt_from_particle_set

<HEP common: procedures>+≡
  subroutine hepevt_from_particle_set (particle_set, keep_beams)
    type(particle_set_t), intent(in), target :: particle_set
    type(particle_t), pointer :: prt
    type(particle_set_t), target :: pset_hepevt
    logical, intent(in), optional :: keep_beams

```

```

integer :: i
call particle_set_to_hepevt_form (particle_set, pset_hepevt, keep_beams)
call hepevt_init (pset_hepevt%n_tot, pset_hepevt%n_out)
do i = 1, pset_hepevt%n_tot
  prt => pset_hepevt%prt(i)
  call hepevt_set_particle (i, &
    particle_get_pdg (prt), &
    particle_get_status (prt), &
    particle_get_parents (prt), &
    particle_get_children (prt), &
    particle_get_momentum (prt), &
    particle_get_p2 (prt), &
    particle_get_helicity (prt))
end do
call particle_set_final (pset_hepevt)
end subroutine hepevt_from_particle_set

```

#### 17.4.11 Data Transfer: events

```

<HEP common: public>+≡
  public :: hepeup_from_event

<HEP common: procedures>+≡
  subroutine hepeup_from_event (event, keep_beams, process_index)
    type(event_t), intent(in), target :: event
    logical, intent(in), optional :: keep_beams
    integer, intent(in), optional :: process_index
    type(particle_set_t), pointer :: particle_set
    real(default) :: scale, alpha_qcd
    if (event%has_particle_set ()) then
      particle_set => event%get_particle_set_ptr ()
      call hepeup_from_particle_set (particle_set, keep_beams)
      if (present (process_index)) &
        call hepeup_set_event_parameters (proc_id = process_index)
      scale = event%get_fac_scale ()
      if (scale /= 0) &
        call hepeup_set_event_parameters (scale = scale)
      alpha_qcd = event%get_alpha_s ()
      if (alpha_qcd /= 0) &
        call hepeup_set_event_parameters (alpha_qcd = alpha_qcd)
      call hepeup_set_event_parameters (weight = event%get_weight ())
    else
      call msg_bug ("HEPEUP: event incomplete")
    end if
  end subroutine hepeup_from_event

```

Fill the HEPEVT (event) common block:

```

<HEP common: public>+≡
  public :: hepevt_from_event

<HEP common: procedures>+≡
  subroutine hepevt_from_event (event, i_evt, keep_beams)
    type(event_t), intent(in), target :: event

```

```

integer, intent(in), optional :: i_evt
logical, intent(in), optional :: keep_beams
type(particle_set_t), pointer :: particle_set
if (event%has_particle_set ()) then
    particle_set => event%get_particle_set_ptr ()
    call hepevt_from_particle_set (particle_set, keep_beams)
    call hepevt_set_event_parameters ( &
        weight = event%get_weight (), &
        function_value = event%get_sqme ())
    if (present (i_evt)) &
        call hepevt_set_event_parameters (i_evt = i_evt)
else
    call msg_bug ("HEPEVT: event incomplete")
end if
end subroutine hepevt_from_event

```

## 17.5 LHEF Input/Output

The LHEF event record is standardized. It is an ASCII format. We try our best at using it for both input and output.

`<eio_lhef.f90>`≡  
*<File header>*

module eio\_lhef

```

    use kinds !NODEP!
<Use file utils>
<Use strings>
    use diagnostics !NODEP!
    use unit_tests

```

```

    use lorentz !NODEP!
    use particles
    use beams
    use processes
    use events
    use eio_data
    use eio_base
    use hep_common

```

*<Standard module head>*

*<EIO LHEF: public>*

*<EIO LHEF: types>*

contains

*<EIO LHEF: procedures>*

*<EIO LHEF: tests>*

```
end module eio_lhef
```

### 17.5.1 Type

```
<EIO LHEF: public>≡
    public :: eio_lhef_t
<EIO LHEF: types>≡
    type, extends (eio_t) :: eio_lhef_t
        type(string_t) :: filename
        logical :: writing = .false.
        integer :: unit = 0
        logical :: keep_beams = .false.
    contains
        <EIO LHEF: eio_lhef: TBP>
    end type eio_lhef_t
```

### 17.5.2 Specific Methods

Set parameters that are specifically used with LHEF.

```
<EIO LHEF: eio_lhef: TBP>≡
    procedure :: set_parameters => eio_lhef_set_parameters
<EIO LHEF: procedures>≡
    subroutine eio_lhef_set_parameters (eio, keep_beams)
        class(eio_lhef_t), intent(inout) :: eio
        logical, intent(in), optional :: keep_beams
        if (present (keep_beams)) eio%keep_beams = keep_beams
    end subroutine eio_lhef_set_parameters
```

### 17.5.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

```
<EIO LHEF: eio_lhef: TBP>+≡
    procedure :: write => eio_lhef_write
<EIO LHEF: procedures>+≡
    subroutine eio_lhef_write (object, unit)
        class(eio_lhef_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        write (u, "(1x,A)") "LHEF event stream:"
        if (object%writing) then
            write (u, "(3x,A,A)") "Writing to file   = ", char (object%filename)
        else
            write (u, "(3x,A)") "[closed]"
        end if
        write (u, "(3x,A,L1)") "Keep beams      = ", object%keep_beams
    end subroutine eio_lhef_write
```

Finalizer: close any open file.

```

<EIO LHEF: eio_lhef: TBP>+≡
  procedure :: final => eio_lhef_final

<EIO LHEF: procedures>+≡
  subroutine eio_lhef_final (object)
    class(eio_lhef_t), intent(inout) :: object
    if (object%writing) then
      write (msg_buffer, "(A,A,A)") "Events: closing LHEF file '", &
        char (object%filename), "'"
      call msg_message ()
      call lhef_write_footer (object%unit)
      close (object%unit)
      object%writing = .false.
    end if
  end subroutine eio_lhef_final

```

Initialize event writing.

```

<EIO LHEF: eio_lhef: TBP>+≡
  procedure :: init_out => eio_lhef_init_out

<EIO LHEF: procedures>+≡
  subroutine eio_lhef_init_out (eio, sample, process_ptr, data, success)
    class(eio_lhef_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    integer :: i
    if (.not. present (data)) &
      call msg_bug ("LHEF initialization: missing data")
    if (data%n_beam /= 2) &
      call msg_fatal ("LHEF: defined for scattering processes only")
    eio%filename = sample // ".lhef"
    eio%unit = free_unit ()
    write (msg_buffer, "(A,A,A)") "Events: writing to LHEF file '", &
      char (eio%filename), "'"
    call msg_message ()
    eio%writing = .true.
    open (eio%unit, file = char (eio%filename), &
      action = "write", status = "replace")
    call lhef_write_header (eio%unit)
    call heprup_init &
      (data%pdg_beam, &
      data%energy_beam, &
      n_processes = data%n_proc, &
      unweighted = data%unweighted, &
      negative_weights = data%negative_weights)
    do i = 1, data%n_proc
      call heprup_set_process_parameters (i = i, &
        process_id = data%proc_num_id(i), &
        cross_section = data%cross_section(i), &
        error = data%error(i))
    end do
  end subroutine eio_lhef_init_out

```

```

        call heprup_write_lhef (eio%unit)
        if (present (success)) success = .true.
    end subroutine eio_lhef_init_out

```

Initialize event reading.

```

<EIO LHEF: eio_lhef: TBP>+≡
    procedure :: init_in => eio_lhef_init_in

<EIO LHEF: procedures>+≡
    subroutine eio_lhef_init_in (eio, sample, process_ptr, data, success)
        class(eio_lhef_t), intent(inout) :: eio
        type(string_t), intent(in) :: sample
        type(process_ptr_t), dimension(:), intent(in) :: process_ptr
        type(event_sample_data_t), intent(in), optional :: data
        logical, intent(out), optional :: success
        call msg_bug ("LHEF: event input not supported")
    end subroutine eio_lhef_init_in

```

Switch from input to output: reopen the file for reading.

```

<EIO LHEF: eio_lhef: TBP>+≡
    procedure :: switch_inout => eio_lhef_switch_inout

<EIO LHEF: procedures>+≡
    subroutine eio_lhef_switch_inout (eio, success)
        class(eio_lhef_t), intent(inout) :: eio
        logical, intent(out), optional :: success
        call msg_bug ("LHEF: in-out switch not supported")
    end subroutine eio_lhef_switch_inout

```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

```

<EIO LHEF: eio_lhef: TBP>+≡
    procedure :: output => eio_lhef_output

<EIO LHEF: procedures>+≡
    subroutine eio_lhef_output (eio, event, i_prc)
        class(eio_lhef_t), intent(inout) :: eio
        type(event_t), intent(in), target :: event
        integer, intent(in) :: i_prc
        real(default) :: weight, sqme_evt, sqme_prc
        if (eio%writing) then
            call hepeup_from_event (event, &
                process_index = i_prc, &
                keep_beams = eio%keep_beams)
            call hepeup_write_lhef (eio%unit)
        else
            call eio%write ()
            call msg_fatal ("LHEF file is not open for writing")
        end if
    end subroutine eio_lhef_output

```

Input an event.

```

<EIO LHEF: eio_lhef: TBP>+≡
  procedure :: input_i_prc => eio_lhef_input_i_prc
  procedure :: input_event => eio_lhef_input_event

<EIO LHEF: procedures>+≡
  subroutine eio_lhef_input_i_prc (eio, i_prc, iostat)
    class(eio_lhef_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    call msg_bug ("LHEF: event input not supported")
  end subroutine eio_lhef_input_i_prc

  subroutine eio_lhef_input_event (eio, event, iostat)
    class(eio_lhef_t), intent(inout) :: eio
    type(event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    call msg_bug ("LHEF: event input not supported")
  end subroutine eio_lhef_input_event

```

#### 17.5.4 Les Houches Event File: header/footer

These two routines write the header and footer for the Les Houches Event File format (LHEF).

The current version writes no information except for the generator name and version.

```

<EIO LHEF: procedures>+≡
  subroutine lhef_write_header (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, '(A)') '<LesHouchesEvents version="1.0">'
    write (u, '(A)') '<header>'
    write (u, '(A)') ' <generator_name>WHIZARD</generator_name>'
    write (u, '(A)') ' <generator_version><Version></generator_version>'
    write (u, '(A)') '</header>'
  end subroutine lhef_write_header

  subroutine lhef_write_footer (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit); if (u < 0) return
    write (u, '(A)') '</LesHouchesEvents>'
  end subroutine lhef_write_footer

```

#### 17.5.5 Unit tests

```

<EIO LHEF: public>+≡
  public :: eio_lhef_test

```

```

<EIO LHEF: tests>≡
  subroutine eio_lhef_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <EIO LHEF: execute tests>
  end subroutine eio_lhef_test

```

## Test I/O methods

We test the implementation of all I/O methods.

```

<EIO LHEF: execute tests>≡
  call test (eio_lhef_1, "eio_lhef_1", &
    "read and write event contents", &
    u, results)

<EIO LHEF: tests>+≡
  subroutine eio_lhef_1 (u)
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_ptr_t) :: process_ptr
    type(process_instance_t), allocatable, target :: process_instance
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_lhef_1"
    write (u, "(A)")  "* Purpose: generate an event and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    allocate (process)
    process_ptr%ptr => process
    allocate (process_instance)
    call prepare_test_process (process, process_instance)
    call process_instance%setup_event_data ()

    allocate (event)
    call event%connect (process_instance)

    call data%init (1)
    data%n_beam = 2
    data%pdg_beam = 25
    data%energy_beam = 500
    data%cross_section(1) = 100
    data%error(1) = 1

    write (u, "(A)")
    write (u, "(A)")  "* Generate and write an event"
    write (u, "(A)")

```



```

sample = "eio_lhef_1"

allocate (eio_lhef_t :: eio)

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

open (u_file, file = "eio_lhef_1.lhef", &
      action = "read", status = "old")
do
  read (u_file, "(A)", iostat = iostat) buffer
  if (buffer(1:21) == " <generator_version>") buffer = "[...]"
  if (iostat /= 0) exit
  write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_lhef_t :: eio)

select type (eio)
type is (eio_lhef_t)
  call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_lhef_1"

end subroutine eio_lhef_1

```

## 17.6 HepMC events

This section provides the interface to the HepMC C++ library for handling Monte-Carlo events.

Each C++ class of HepMC that we use is mirrored by a Fortran type, which contains as its only component the C pointer to the C++ object.

Each C++ method of HepMC that we use has a C wrapper function. This function takes a pointer to the host object as its first argument. Further arguments are either C pointers, or in the case of simple types (integer, real), interoperable C/Fortran objects.

The C wrapper functions have explicit interfaces in the Fortran module. They are called by Fortran wrapper procedures. These are treated as methods of the corresponding Fortran type.

```
(hepmc_interface.f90)≡  
  <File header>  
  
  module hepmc_interface  
  
    use iso_c_binding !NODEP!  
    <Use kinds>  
    <Use strings>  
    use constants !NODEP!  
    use lorentz !NODEP!  
    use models  
    use flavors  
    use colors  
    use helicities  
    use quantum_numbers  
    use polarizations  
  
    <Standard module head>  
  
    <HepMC interface: public>  
  
    <HepMC interface: types>  
  
    <HepMC interface: interfaces>  
  
    contains  
  
    <HepMC interface: procedures>  
  
  end module hepmc_interface
```

### 17.6.1 Interface check

This function can be called in order to verify that we are using the actual HepMC library, and not the dummy version.

```
<HepMC interface: interfaces>≡  
  interface  
    logical(c_bool) function hepmc_available () bind(C)  
    import
```

```

        end function hepmc_available
    end interface
    <HepMC interface: public>≡
        public :: hepmc_is_available
    <HepMC interface: procedures>≡
        function hepmc_is_available () result (flag)
            logical :: flag
            flag = hepmc_available ()
        end function hepmc_is_available

```

### 17.6.2 FourVector

The C version of four-vectors is often transferred by value, and the associated procedures are all inlined. The wrapper needs to transfer by reference, so we create FourVector objects on the heap which have to be deleted explicitly. The input is a `vector4_t` or `vector3_t` object from the `lorentz` module.

```

    <HepMC interface: public>+≡
        public :: hepmc_four_vector_t
    <HepMC interface: types>≡
        type :: hepmc_four_vector_t
        private
        type(c_ptr) :: obj
    end type hepmc_four_vector_t

```

In the C constructor, the zero-component (fourth argument) is optional; if missing, it is set to zero. The Fortran version has initializer form and takes either a three-vector or a four-vector. A further version extracts the four-vector from a HepMC particle object.

```

    <HepMC interface: interfaces>+≡
        interface
            type(c_ptr) function new_four_vector_xyz (x, y, z) bind(C)
                import
                real(c_double), value :: x, y, z
            end function new_four_vector_xyz
        end interface
        interface
            type(c_ptr) function new_four_vector_xyz_t (x, y, z, t) bind(C)
                import
                real(c_double), value :: x, y, z, t
            end function new_four_vector_xyz_t
        end interface
    <HepMC interface: public>+≡
        public :: hepmc_four_vector_init
    <HepMC interface: interfaces>+≡
        interface hepmc_four_vector_init
            module procedure hepmc_four_vector_init_v4
            module procedure hepmc_four_vector_init_v3
            module procedure hepmc_four_vector_init_hepmc_prt
        end interface

```

```

<HepMC interface: procedures>+≡
subroutine hepmc_four_vector_init_v4 (pp, p)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(vector4_t), intent(in) :: p
  real(default), dimension(0:3) :: pa
  pa = vector4_get_components (p)
  pp%obj = new_four_vector_xyz &
    (real (pa(1), c_double), &
     real (pa(2), c_double), &
     real (pa(3), c_double), &
     real (pa(0), c_double))
end subroutine hepmc_four_vector_init_v4

subroutine hepmc_four_vector_init_v3 (pp, p)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(vector3_t), intent(in) :: p
  real(default), dimension(3) :: pa
  pa = vector3_get_components (p)
  pp%obj = new_four_vector_xyz &
    (real (pa(1), c_double), &
     real (pa(2), c_double), &
     real (pa(3), c_double))
end subroutine hepmc_four_vector_init_v3

subroutine hepmc_four_vector_init_hepmc_prt (pp, prt)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(hepmc_particle_t), intent(in) :: prt
  pp%obj = gen_particle_momentum (prt%obj)
end subroutine hepmc_four_vector_init_hepmc_prt

```

Here, the destructor is explicitly needed.

```

<HepMC interface: interfaces>+≡
interface
  subroutine four_vector_delete (p_obj) bind(C)
    import
    type(c_ptr), value :: p_obj
  end subroutine four_vector_delete
end interface

<HepMC interface: public>+≡
public :: hepmc_four_vector_final

<HepMC interface: procedures>+≡
subroutine hepmc_four_vector_final (p)
  type(hepmc_four_vector_t), intent(inout) :: p
  call four_vector_delete (p%obj)
end subroutine hepmc_four_vector_final

```

Convert to a Lorentz vector.

```

<HepMC interface: interfaces>+≡
interface
  function four_vector_px (p_obj) result (px) bind(C)
    import
    real(c_double) :: px

```

```

        type(c_ptr), value :: p_obj
    end function four_vector_px
end interface
interface
    function four_vector_py (p_obj) result (py) bind(C)
        import
        real(c_double) :: py
        type(c_ptr), value :: p_obj
    end function four_vector_py
end interface
interface
    function four_vector_pz (p_obj) result (pz) bind(C)
        import
        real(c_double) :: pz
        type(c_ptr), value :: p_obj
    end function four_vector_pz
end interface
interface
    function four_vector_e (p_obj) result (e) bind(C)
        import
        real(c_double) :: e
        type(c_ptr), value :: p_obj
    end function four_vector_e
end interface
<HepMC interface: public>+≡
    public :: hepmc_four_vector_to_vector4
<HepMC interface: procedures>+≡
    subroutine hepmc_four_vector_to_vector4 (pp, p)
        type(hepmc_four_vector_t), intent(in) :: pp
        type(vector4_t), intent(out) :: p
        real(default) :: E
        real(default), dimension(3) :: p3
        E = four_vector_e (pp%obj)
        p3(1) = four_vector_px (pp%obj)
        p3(2) = four_vector_py (pp%obj)
        p3(3) = four_vector_pz (pp%obj)
        p = vector4_moving (E, vector3_moving (p3))
    end subroutine hepmc_four_vector_to_vector4

```

### 17.6.3 Polarization

Polarization objects are temporarily used for assigning particle polarization. We add a flag `polarized`. If this is false, the polarization is not set and should not be transferred to `hepmc_particle` objects.

```

<HepMC interface: public>+≡
    public :: hepmc_polarization_t
<HepMC interface: types>+≡
    type :: hepmc_polarization_t
    private
    logical :: polarized = .false.
    type(c_ptr) :: obj

```

```
end type hepmc_polarization_t
```

Constructor. The C wrapper takes polar and azimuthal angle as arguments. The Fortran version allows for either a complete polarization density matrix, or for a definite (diagonal) helicity.

*HepMC does not allow to specify the degree of polarization, therefore we have to map it to either 0 or 1. We choose 0 for polarization less than 0.5 and 1 for polarization greater than 0.5. Even this simplification works only for spin-1/2 and for massless particles; massive vector bosons cannot be treated this way. In particular, zero helicity is always translated as unpolarized.*

*(HepMC interface: interfaces)+≡*

```
interface
  type(c_ptr) function new_polarization (theta, phi) bind(C)
  import
    real(c_double), value :: theta, phi
  end function new_polarization
end interface
```

*(HepMC interface: public)+≡*

```
public :: hepmc_polarization_init
```

*(HepMC interface: interfaces)+≡*

```
interface hepmc_polarization_init
  module procedure hepmc_polarization_init_pol
  module procedure hepmc_polarization_init_hel
  module procedure hepmc_polarization_init_int
end interface
```

*(HepMC interface: procedures)+≡*

```
subroutine hepmc_polarization_init_pol (hpol, pol)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(polarization_t), intent(in) :: pol
  real(default) :: r, theta, phi
  if (polarization_is_polarized (pol)) then
    call polarization_to_angles (pol, r, theta, phi)
    if (r >= 0.5) then
      hpol%polarized = .true.
      hpol%obj = new_polarization &
        (real (theta, c_double), real (phi, c_double))
    end if
  end if
end subroutine hepmc_polarization_init_pol

subroutine hepmc_polarization_init_hel (hpol, hel)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(helicity_t), intent(in) :: hel
  integer, dimension(2) :: h
  if (helicity_is_defined (hel)) then
    h = helicity_get (hel)
    select case (h(1))
    case (1:)
      hpol%polarized = .true.
      hpol%obj = new_polarization (0._c_double, 0._c_double)
    case (:-1)
      hpol%polarized = .true.
```

```

        hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
    end select
end if
end subroutine hepmc_polarization_init_hel

subroutine hepmc_polarization_init_int (hpol, hel)
    type(hepmc_polarization_t), intent(out) :: hpol
    integer, intent(in) :: hel
    select case (hel)
    case (1:)
        hpol%polarized = .true.
        hpol%obj = new_polarization (0._c_double, 0._c_double)
    case (:-1)
        hpol%polarized = .true.
        hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
    end select
end subroutine hepmc_polarization_init_int

```

Destructor. The C object is deallocated only if the polarized flag is set.

```

<HepMC interface: interfaces>+≡
    interface
        subroutine polarization_delete (pol_obj) bind(C)
            import
            type(c_ptr), value :: pol_obj
        end subroutine polarization_delete
    end interface

<HepMC interface: public>+≡
    public :: hepmc_polarization_final

<HepMC interface: procedures>+≡
    subroutine hepmc_polarization_final (hpol)
        type(hepmc_polarization_t), intent(inout) :: hpol
        if (hpol%polarized) call polarization_delete (hpol%obj)
    end subroutine hepmc_polarization_final

```

Recover polarization from HepMC polarization object (with the abovementioned deficiencies).

```

<HepMC interface: interfaces>+≡
    interface
        function polarization_theta (pol_obj) result (theta) bind(C)
            import
            real(c_double) :: theta
            type(c_ptr), value :: pol_obj
        end function polarization_theta
    end interface

    interface
        function polarization_phi (pol_obj) result (phi) bind(C)
            import
            real(c_double) :: phi
            type(c_ptr), value :: pol_obj
        end function polarization_phi
    end interface

```

```

<HepMC interface: public>+≡
    public :: hepmc_polarization_to_pol

<HepMC interface: procedures>+≡
    subroutine hepmc_polarization_to_pol (hpol, flv, pol)
        type(hepmc_polarization_t), intent(in) :: hpol
        type(flavor_t), intent(in) :: flv
        type(polarization_t), intent(out) :: pol
        real(default) :: theta, phi
        theta = polarization_theta (hpol%obj)
        phi = polarization_phi (hpol%obj)
        call polarization_init_angles (pol, flv, 1._default, theta, phi)
    end subroutine hepmc_polarization_to_pol

```

Recover helicity. Here,  $\phi$  is ignored and only the sign of  $\cos \theta$  is relevant, mapped to positive/negative helicity.

```

<HepMC interface: public>+≡
    public :: hepmc_polarization_to_hel

<HepMC interface: procedures>+≡
    subroutine hepmc_polarization_to_hel (hpol, flv, hel)
        type(hepmc_polarization_t), intent(in) :: hpol
        type(flavor_t), intent(in) :: flv
        type(helicity_t), intent(out) :: hel
        real(default) :: theta
        integer :: hmax
        theta = polarization_theta (hpol%obj)
        hmax = flavor_get_spin_type (flv) / 2
        call helicity_init (hel, sign (hmax, nint (cos (theta))))
    end subroutine hepmc_polarization_to_hel

```

#### 17.6.4 GenParticle

Particle objects have the obvious meaning.

```

<HepMC interface: public>+≡
    public :: hepmc_particle_t

<HepMC interface: types>+≡
    type :: hepmc_particle_t
        private
        type(c_ptr) :: obj
    end type hepmc_particle_t

```

Constructor. The C version takes a FourVector object, which in the Fortran wrapper is created on the fly from a `vector4` Lorentz vector.

No destructor is needed as long as all particles are entered into vertex containers.

```

<HepMC interface: interfaces>+≡
    interface
        type(c_ptr) function new_gen_particle (prt_obj, pdg_id, status) bind(C)
            import
            type(c_ptr), value :: prt_obj

```



```

        integer(c_int), value :: pdg_id, status
    end function new_gen_particle
end interface

<HepMC interface: public>+≡
    public :: hepmc_particle_init

<HepMC interface: procedures>+≡
    subroutine hepmc_particle_init (prt, p, pdg, status)
        type(hepmc_particle_t), intent(out) :: prt
        type(vector4_t), intent(in) :: p
        integer, intent(in) :: pdg, status
        type(hepmc_four_vector_t) :: pp
        call hepmc_four_vector_init (pp, p)
        prt%obj = new_gen_particle (pp%obj, int (pdg, c_int), int (status, c_int))
        call hepmc_four_vector_final (pp)
    end subroutine hepmc_particle_init

```

Set the particle color flow.

```

<HepMC interface: interfaces>+≡
    interface
        subroutine gen_particle_set_flow (prt_obj, code_index, code) bind(C)
            import
            type(c_ptr), value :: prt_obj
            integer(c_int), value :: code_index, code
        end subroutine gen_particle_set_flow
    end interface

```

Set the particle color. Either from a color\_t object or directly from a pair of integers.

```

<HepMC interface: interfaces>+≡
    interface hepmc_particle_set_color
        module procedure hepmc_particle_set_color_col
        module procedure hepmc_particle_set_color_int
    end interface hepmc_particle_set_color

<HepMC interface: public>+≡
    public :: hepmc_particle_set_color

<HepMC interface: procedures>+≡
    subroutine hepmc_particle_set_color_col (prt, col)
        type(hepmc_particle_t), intent(inout) :: prt
        type(color_t), intent(in) :: col
        integer(c_int) :: c
        c = color_get_col (col)
        if (c /= 0) call gen_particle_set_flow (prt%obj, 1_c_int, c)
        c = color_get_acl (col)
        if (c /= 0) call gen_particle_set_flow (prt%obj, 2_c_int, c)
    end subroutine hepmc_particle_set_color_col

    subroutine hepmc_particle_set_color_int (prt, col)
        type(hepmc_particle_t), intent(inout) :: prt
        integer, dimension(2), intent(in) :: col
        integer(c_int) :: c
        c = col(1)
        if (c /= 0) call gen_particle_set_flow (prt%obj, 1_c_int, c)
    end subroutine hepmc_particle_set_color_int

```

```

c = col(2)
if (c /= 0) call gen_particle_set_flow (prt%obj, 2_c_int, c)
end subroutine hepmc_particle_set_color_int

```

Set the particle polarization. For the restrictions on particle polarization in HepMC, see above `hepmc_polarization_init`.

*(HepMC interface: interfaces)+≡*

```

interface
  subroutine gen_particle_set_polarization (prt_obj, pol_obj) bind(C)
  import
  type(c_ptr), value :: prt_obj, pol_obj
  end subroutine gen_particle_set_polarization
end interface

```

*(HepMC interface: public)+≡*

```

public :: hepmc_particle_set_polarization

```

*(HepMC interface: interfaces)+≡*

```

interface hepmc_particle_set_polarization
  module procedure hepmc_particle_set_polarization_pol
  module procedure hepmc_particle_set_polarization_hel
  module procedure hepmc_particle_set_polarization_int
end interface

```

*(HepMC interface: procedures)+≡*

```

subroutine hepmc_particle_set_polarization_pol (prt, pol)
  type(hepmc_particle_t), intent(inout) :: prt
  type(polarization_t), intent(in) :: pol
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, pol)
  if (hpol%polarized) call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_pol

```

```

subroutine hepmc_particle_set_polarization_hel (prt, hel)
  type(hepmc_particle_t), intent(inout) :: prt
  type(helicity_t), intent(in) :: hel
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, hel)
  if (hpol%polarized) call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_hel

```

```

subroutine hepmc_particle_set_polarization_int (prt, hel)
  type(hepmc_particle_t), intent(inout) :: prt
  integer, intent(in) :: hel
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, hel)
  if (hpol%polarized) call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_int

```

Return the HepMC barcode (unique integer ID) of the particle.

*(HepMC interface: interfaces)+≡*

```

interface
  function gen_particle_barcode (prt_obj) result (barcode) bind(C)
  import
    integer(c_int) :: barcode
    type(c_ptr), value :: prt_obj
  end function gen_particle_barcode
end interface

<HepMC interface: public>+≡
  public :: hepmc_particle_get_barcode

<HepMC interface: procedures>+≡
  function hepmc_particle_get_barcode (prt) result (barcode)
  integer :: barcode
  type(hepmc_particle_t), intent(in) :: prt
  barcode = gen_particle_barcode (prt%obj)
end function hepmc_particle_get_barcode

```

Return the four-vector component of the particle object as a `vector4_t` Lorentz vector.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function gen_particle_momentum (prt_obj) bind(C)
    import
      type(c_ptr), value :: prt_obj
    end function gen_particle_momentum
  end interface

<HepMC interface: public>+≡
  public :: hepmc_particle_get_momentum

<HepMC interface: procedures>+≡
  function hepmc_particle_get_momentum (prt) result (p)
  type(vector4_t) :: p
  type(hepmc_particle_t), intent(in) :: prt
  type(hepmc_four_vector_t) :: pp
  call hepmc_four_vector_init (pp, prt)
  call hepmc_four_vector_to_vector4 (pp, p)
  call hepmc_four_vector_final (pp)
end function hepmc_particle_get_momentum

```

Return the invariant mass squared of the particle object. HepMC stores the signed invariant mass (no squaring).

```

<HepMC interface: interfaces>+≡
  interface
    function gen_particle_generated_mass (prt_obj) result (mass) bind(C)
    import
      real(c_double) :: mass
      type(c_ptr), value :: prt_obj
    end function gen_particle_generated_mass
  end interface

<HepMC interface: public>+≡
  public :: hepmc_particle_get_mass_squared

```

```

<HepMC interface: procedures>+≡
function hepmc_particle_get_mass_squared (prt) result (m2)
    real(default) :: m2
    type(hepmc_particle_t), intent(in) :: prt
    real(default) :: m
    m = gen_particle_generated_mass (prt%obj)
    m2 = sign (m**2, m)
end function hepmc_particle_get_mass_squared

```

Return the PDG ID:

```

<HepMC interface: interfaces>+≡
interface
    function gen_particle_pdg_id (prt_obj) result (pdg_id) bind(C)
        import
        integer(c_int) :: pdg_id
        type(c_ptr), value :: prt_obj
    end function gen_particle_pdg_id
end interface

<HepMC interface: public>+≡
public :: hepmc_particle_get_pdg

<HepMC interface: procedures>+≡
function hepmc_particle_get_pdg (prt) result (pdg)
    integer :: pdg
    type(hepmc_particle_t), intent(in) :: prt
    pdg = gen_particle_pdg_id (prt%obj)
end function hepmc_particle_get_pdg

```

Return the status code:

```

<HepMC interface: interfaces>+≡
interface
    function gen_particle_status (prt_obj) result (status) bind(C)
        import
        integer(c_int) :: status
        type(c_ptr), value :: prt_obj
    end function gen_particle_status
end interface

<HepMC interface: public>+≡
public :: hepmc_particle_get_status

<HepMC interface: procedures>+≡
function hepmc_particle_get_status (prt) result (status)
    integer :: status
    type(hepmc_particle_t), intent(in) :: prt
    status = gen_particle_status (prt%obj)
end function hepmc_particle_get_status

```

Return the production/decay vertex (as a pointer, no finalization necessary).

```

<HepMC interface: interfaces>+≡
interface
    type(c_ptr) function gen_particle_production_vertex (prt_obj) bind(C)
        import

```

```

        type(c_ptr), value :: prt_obj
    end function gen_particle_production_vertex
end interface
interface
    type(c_ptr) function gen_particle_end_vertex (prt_obj) bind(C)
        import
        type(c_ptr), value :: prt_obj
    end function gen_particle_end_vertex
end interface

<HepMC interface: public>+≡
    public :: hePMC_particle_get_production_vertex
    public :: hePMC_particle_get_decay_vertex

<HepMC interface: procedures>+≡
    function hePMC_particle_get_production_vertex (prt) result (v)
        type(hePMC_vertex_t) :: v
        type(hePMC_particle_t), intent(in) :: prt
        v%obj = gen_particle_production_vertex (prt%obj)
    end function hePMC_particle_get_production_vertex

    function hePMC_particle_get_decay_vertex (prt) result (v)
        type(hePMC_vertex_t) :: v
        type(hePMC_particle_t), intent(in) :: prt
        v%obj = gen_particle_end_vertex (prt%obj)
    end function hePMC_particle_get_decay_vertex

Return the number of parents/children.

<HepMC interface: public>+≡
    public :: hePMC_particle_get_n_parents
    public :: hePMC_particle_get_n_children

<HepMC interface: procedures>+≡
    function hePMC_particle_get_n_parents (prt) result (n_parents)
        integer :: n_parents
        type(hePMC_particle_t), intent(in) :: prt
        type(hePMC_vertex_t) :: v
        v = hePMC_particle_get_production_vertex (prt)
        if (hePMC_vertex_is_valid (v)) then
            n_parents = hePMC_vertex_get_n_in (v)
        else
            n_parents = 0
        end if
    end function hePMC_particle_get_n_parents

    function hePMC_particle_get_n_children (prt) result (n_children)
        integer :: n_children
        type(hePMC_particle_t), intent(in) :: prt
        type(hePMC_vertex_t) :: v
        v = hePMC_particle_get_decay_vertex (prt)
        if (hePMC_vertex_is_valid (v)) then
            n_children = hePMC_vertex_get_n_out (v)
        else
            n_children = 0
        end if
    end function hePMC_particle_get_n_children

```

```
end function hepmc_particle_get_n_children
```

Convenience function: Return the array of parent particles for a given HepMC particle. The contents are HepMC barcodes that still have to be mapped to the particle indices.

```
<HepMC interface: public>+≡
```

```
public :: hepmc_particle_get_parent_barcodes
public :: hepmc_particle_get_child_barcodes
```

```
<HepMC interface: procedures>+≡
```

```
function hepmc_particle_get_parent_barcodes (prt) result (parent_barcode)
  type(hepmc_particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: parent_barcode
  type(hepmc_vertex_t) :: v
  type(hepmc_vertex_particle_in_iterator_t) :: it
  integer :: i
  v = hepmc_particle_get_production_vertex (prt)
  if (hepmc_vertex_is_valid (v)) then
    allocate (parent_barcode (hepmc_vertex_get_n_in (v)))
    if (size (parent_barcode) /= 0) then
      call hepmc_vertex_particle_in_iterator_init (it, v)
      do i = 1, size (parent_barcode)
        parent_barcode(i) = hepmc_particle_get_barcode &
          (hepmc_vertex_particle_in_iterator_get (it))
        call hepmc_vertex_particle_in_iterator_advance (it)
      end do
      call hepmc_vertex_particle_in_iterator_final (it)
    end if
  else
    allocate (parent_barcode (0))
  end if
end function hepmc_particle_get_parent_barcodes
```

```
function hepmc_particle_get_child_barcodes (prt) result (child_barcode)
  type(hepmc_particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: child_barcode
  type(hepmc_vertex_t) :: v
  type(hepmc_vertex_particle_out_iterator_t) :: it
  integer :: i
  v = hepmc_particle_get_decay_vertex (prt)
  if (hepmc_vertex_is_valid (v)) then
    allocate (child_barcode (hepmc_vertex_get_n_out (v)))
    call hepmc_vertex_particle_out_iterator_init (it, v)
    if (size (child_barcode) /= 0) then
      do i = 1, size (child_barcode)
        child_barcode(i) = hepmc_particle_get_barcode &
          (hepmc_vertex_particle_out_iterator_get (it))
        call hepmc_vertex_particle_out_iterator_advance (it)
      end do
      call hepmc_vertex_particle_out_iterator_final (it)
    end if
  else
    allocate (child_barcode (0))
  end if
end function
```

```
end function hepmc_particle_get_child_barcodes
```

Return the polarization (assuming that the particle is completely polarized).  
Note that the generated polarization object needs finalization.

```
<HepMC interface: interfaces>+≡
interface
    type(c_ptr) function gen_particle_polarization (prt_obj) bind(C)
    import
    type(c_ptr), value :: prt_obj
end function gen_particle_polarization
end interface

<HepMC interface: public>+≡
public :: hepmc_particle_get_polarization

<HepMC interface: procedures>+≡
function hepmc_particle_get_polarization (prt) result (pol)
    type(hepmc_polarization_t) :: pol
    type(hepmc_particle_t), intent(in) :: prt
    pol%obj = gen_particle_polarization (prt%obj)
end function hepmc_particle_get_polarization
```

Return the particle color as a two-dimensional array (color, anticolor).

```
<HepMC interface: interfaces>+≡
interface
    function gen_particle_flow (prt_obj, code_index) result (code) bind(C)
    import
    integer(c_int) :: code
    type(c_ptr), value :: prt_obj
    integer(c_int), value :: code_index
end function gen_particle_flow
end interface

<HepMC interface: public>+≡
public :: hepmc_particle_get_color

<HepMC interface: procedures>+≡
function hepmc_particle_get_color (prt) result (col)
    integer, dimension(2) :: col
    type(hepmc_particle_t), intent(in) :: prt
    col(1) = gen_particle_flow (prt%obj, 1)
    col(2) = - gen_particle_flow (prt%obj, 2)
end function hepmc_particle_get_color
```

### 17.6.5 GenVertex

Vertices are made of particles (incoming and outgoing).

```
<HepMC interface: public>+≡
public :: hepmc_vertex_t
```

```

<HepMC interface: types>+≡
  type :: hepmc_vertex_t
  private
  type(c_ptr) :: obj
end type hepmc_vertex_t

```

Constructor. Two versions, one plain, one with the position in space and time (measured in mm) as argument. The Fortran version has initializer form, and the vertex position is an optional argument.

A destructor is unnecessary as long as all vertices are entered into an event container.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function new_gen_vertex () bind(C)
    import
    end function new_gen_vertex
  end interface
  interface
    type(c_ptr) function new_gen_vertex_pos (prt_obj) bind(C)
    import
    type(c_ptr), value :: prt_obj
    end function new_gen_vertex_pos
  end interface

```

```

<HepMC interface: public>+≡
  public :: hepmc_vertex_init

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_vertex_init (v, x)
    type(hepmc_vertex_t), intent(out) :: v
    type(vector4_t), intent(in), optional :: x
    type(hepmc_four_vector_t) :: pos
    if (present (x)) then
      call hepmc_four_vector_init (pos, x)
      v%obj = new_gen_vertex_pos (pos%obj)
      call hepmc_four_vector_final (pos)
    else
      v%obj = new_gen_vertex ()
    end if
  end subroutine hepmc_vertex_init

```

Return true if the vertex pointer is non-null:

```

<HepMC interface: interfaces>+≡
  interface
    function gen_vertex_is_valid (v_obj) result (flag) bind(C)
    import
    logical(c_bool) :: flag
    type(c_ptr), value :: v_obj
    end function gen_vertex_is_valid
  end interface
<HepMC interface: public>+≡
  public :: hepmc_vertex_is_valid

```



```

<HepMC interface: procedures>+≡
  function hepmc_vertex_is_valid (v) result (flag)
    logical :: flag
    type(hepmc_vertex_t), intent(in) :: v
    flag = gen_vertex_is_valid (v%obj)
  end function hepmc_vertex_is_valid

```

Add a particle to a vertex, incoming or outgoing.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine gen_vertex_add_particle_in (v_obj, prt_obj) bind(C)
      import
      type(c_ptr), value :: v_obj, prt_obj
    end subroutine gen_vertex_add_particle_in
  end interface
  interface
    subroutine gen_vertex_add_particle_out (v_obj, prt_obj) bind(C)
      import
      type(c_ptr), value :: v_obj, prt_obj
    end subroutine gen_vertex_add_particle_out
  end interface

```

```

<HepMC interface: public>+≡
  public :: hepmc_vertex_add_particle_in
  public :: hepmc_vertex_add_particle_out

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_vertex_add_particle_in (v, prt)
    type(hepmc_vertex_t), intent(inout) :: v
    type(hepmc_particle_t), intent(in) :: prt
    call gen_vertex_add_particle_in (v%obj, prt%obj)
  end subroutine hepmc_vertex_add_particle_in

  subroutine hepmc_vertex_add_particle_out (v, prt)
    type(hepmc_vertex_t), intent(inout) :: v
    type(hepmc_particle_t), intent(in) :: prt
    call gen_vertex_add_particle_out (v%obj, prt%obj)
  end subroutine hepmc_vertex_add_particle_out

```

Return the number of incoming/outgoing particles.

```

<HepMC interface: interfaces>+≡
  interface
    function gen_vertex_particles_in_size (v_obj) result (size) bind(C)
      import
      integer(c_int) :: size
      type(c_ptr), value :: v_obj
    end function gen_vertex_particles_in_size
  end interface
  interface
    function gen_vertex_particles_out_size (v_obj) result (size) bind(C)
      import
      integer(c_int) :: size
      type(c_ptr), value :: v_obj
    end function gen_vertex_particles_out_size
  end interface

```

```

end interface
<HepMC interface: public>+≡
  public :: hepmc_vertex_get_n_in
  public :: hepmc_vertex_get_n_out
<HepMC interface: procedures>+≡
  function hepmc_vertex_get_n_in (v) result (n_in)
    integer :: n_in
    type(hepmc_vertex_t), intent(in) :: v
    n_in = gen_vertex_particles_in_size (v%obj)
  end function hepmc_vertex_get_n_in

  function hepmc_vertex_get_n_out (v) result (n_out)
    integer :: n_out
    type(hepmc_vertex_t), intent(in) :: v
    n_out = gen_vertex_particles_out_size (v%obj)
  end function hepmc_vertex_get_n_out

```

### 17.6.6 Vertex-particle-in iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

```

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_in_iterator_t
<HepMC interface: types>+≡
  type :: hepmc_vertex_particle_in_iterator_t
  private
  type(c_ptr) :: obj
  type(c_ptr) :: v_obj
  end type hepmc_vertex_particle_in_iterator_t

```

Constructor. The iterator is initialized at the first particle in the vertex.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function &
      new_vertex_particles_in_const_iterator (v_obj) bind(C)
    import
    type(c_ptr), value :: v_obj
  end function new_vertex_particles_in_const_iterator
  end interface
<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_in_iterator_init
<HepMC interface: procedures>+≡
  subroutine hepmc_vertex_particle_in_iterator_init (it, v)
    type(hepmc_vertex_particle_in_iterator_t), intent(out) :: it
    type(hepmc_vertex_t), intent(in) :: v
    it%obj = new_vertex_particles_in_const_iterator (v%obj)
    it%v_obj = v%obj
  end subroutine hepmc_vertex_particle_in_iterator_init

```

Destructor. Necessary because the iterator is allocated on the heap.

```
<HepMC interface: interfaces>+≡  
  interface  
    subroutine vertex_particles_in_const_iterator_delete (it_obj) bind(C)  
    import  
    type(c_ptr), value :: it_obj  
    end subroutine vertex_particles_in_const_iterator_delete  
  end interface  
  
<HepMC interface: public>+≡  
  public :: hePMC_vertex_particle_in_iterator_final  
  
<HepMC interface: procedures>+≡  
  subroutine hePMC_vertex_particle_in_iterator_final (it)  
    type(hePMC_vertex_particle_in_iterator_t), intent(inout) :: it  
    call vertex_particles_in_const_iterator_delete (it%obj)  
  end subroutine hePMC_vertex_particle_in_iterator_final
```

Increment

```
<HepMC interface: interfaces>+≡  
  interface  
    subroutine vertex_particles_in_const_iterator_advance (it_obj) bind(C)  
    import  
    type(c_ptr), value :: it_obj  
    end subroutine vertex_particles_in_const_iterator_advance  
  end interface  
  
<HepMC interface: public>+≡  
  public :: hePMC_vertex_particle_in_iterator_advance  
  
<HepMC interface: procedures>+≡  
  subroutine hePMC_vertex_particle_in_iterator_advance (it)  
    type(hePMC_vertex_particle_in_iterator_t), intent(inout) :: it  
    call vertex_particles_in_const_iterator_advance (it%obj)  
  end subroutine hePMC_vertex_particle_in_iterator_advance
```

Reset to the beginning

```
<HepMC interface: interfaces>+≡  
  interface  
    subroutine vertex_particles_in_const_iterator_reset &  
      (it_obj, v_obj) bind(C)  
    import  
    type(c_ptr), value :: it_obj, v_obj  
    end subroutine vertex_particles_in_const_iterator_reset  
  end interface  
  
<HepMC interface: public>+≡  
  public :: hePMC_vertex_particle_in_iterator_reset  
  
<HepMC interface: procedures>+≡  
  subroutine hePMC_vertex_particle_in_iterator_reset (it)  
    type(hePMC_vertex_particle_in_iterator_t), intent(inout) :: it  
    call vertex_particles_in_const_iterator_reset (it%obj, it%v_obj)  
  end subroutine hePMC_vertex_particle_in_iterator_reset
```

Test: return true as long as we are not past the end.

```

<HepMC interface: interfaces>+≡
  interface
    function vertex_particles_in_const_iterator_is_valid &
      (it_obj, v_obj) result (flag) bind(C)
    import
      logical(c_bool) :: flag
      type(c_ptr), value :: it_obj, v_obj
    end function vertex_particles_in_const_iterator_is_valid
  end interface

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_in_iterator_is_valid

<HepMC interface: procedures>+≡
  function hepmc_vertex_particle_in_iterator_is_valid (it) result (flag)
    logical :: flag
    type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
    flag = vertex_particles_in_const_iterator_is_valid (it%obj, it%v_obj)
  end function hepmc_vertex_particle_in_iterator_is_valid

```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function &
      vertex_particles_in_const_iterator_get (it_obj) bind(C)
    import
      type(c_ptr), value :: it_obj
    end function vertex_particles_in_const_iterator_get
  end interface

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_in_iterator_get

<HepMC interface: procedures>+≡
  function hepmc_vertex_particle_in_iterator_get (it) result (prt)
    type(hepmc_particle_t) :: prt
    type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
    prt%obj = vertex_particles_in_const_iterator_get (it%obj)
  end function hepmc_vertex_particle_in_iterator_get

```

### 17.6.7 Vertex-particle-out iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

```

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_out_iterator_t

```

```

<HepMC interface: types>+≡
  type :: hepmc_vertex_particle_out_iterator_t
  private
    type(c_ptr) :: obj
    type(c_ptr) :: v_obj
  end type hepmc_vertex_particle_out_iterator_t

```

Constructor. The iterator is initialized at the first particle in the vertex.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function &
      new_vertex_particles_out_const_iterator (v_obj) bind(C)
    import
      type(c_ptr), value :: v_obj
    end function new_vertex_particles_out_const_iterator
  end interface

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_out_iterator_init

<HepMC interface: procedures>+≡
  subroutine hepmc_vertex_particle_out_iterator_init (it, v)
    type(hepmc_vertex_particle_out_iterator_t), intent(out) :: it
    type(hepmc_vertex_t), intent(in) :: v
    it%obj = new_vertex_particles_out_const_iterator (v%obj)
    it%v_obj = v%obj
  end subroutine hepmc_vertex_particle_out_iterator_init

```

Destructor. Necessary because the iterator is allocated on the heap.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine vertex_particles_out_const_iterator_delete (it_obj) bind(C)
    import
      type(c_ptr), value :: it_obj
    end subroutine vertex_particles_out_const_iterator_delete
  end interface

<HepMC interface: public>+≡
  public :: hepmc_vertex_particle_out_iterator_final

<HepMC interface: procedures>+≡
  subroutine hepmc_vertex_particle_out_iterator_final (it)
    type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
    call vertex_particles_out_const_iterator_delete (it%obj)
  end subroutine hepmc_vertex_particle_out_iterator_final

```

Increment

```

<HepMC interface: interfaces>+≡
  interface
    subroutine vertex_particles_out_const_iterator_advance (it_obj) bind(C)
    import
      type(c_ptr), value :: it_obj
    end subroutine vertex_particles_out_const_iterator_advance
  end interface

```

```

<HepMC interface: public>+≡
    public :: hepmc_vertex_particle_out_iterator_advance

<HepMC interface: procedures>+≡
    subroutine hepmc_vertex_particle_out_iterator_advance (it)
        type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
        call vertex_particles_out_const_iterator_advance (it%obj)
    end subroutine hepmc_vertex_particle_out_iterator_advance

```

Reset to the beginning

```

<HepMC interface: interfaces>+≡
    interface
        subroutine vertex_particles_out_const_iterator_reset &
            (it_obj, v_obj) bind(C)
        import
        type(c_ptr), value :: it_obj, v_obj
        end subroutine vertex_particles_out_const_iterator_reset
    end interface

<HepMC interface: public>+≡
    public :: hepmc_vertex_particle_out_iterator_reset

<HepMC interface: procedures>+≡
    subroutine hepmc_vertex_particle_out_iterator_reset (it)
        type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
        call vertex_particles_out_const_iterator_reset (it%obj, it%v_obj)
    end subroutine hepmc_vertex_particle_out_iterator_reset

```

Test: return true as long as we are not past the end.

```

<HepMC interface: interfaces>+≡
    interface
        function vertex_particles_out_const_iterator_is_valid &
            (it_obj, v_obj) result (flag) bind(C)
        import
        logical(c_bool) :: flag
        type(c_ptr), value :: it_obj, v_obj
        end function vertex_particles_out_const_iterator_is_valid
    end interface

<HepMC interface: public>+≡
    public :: hepmc_vertex_particle_out_iterator_is_valid

<HepMC interface: procedures>+≡
    function hepmc_vertex_particle_out_iterator_is_valid (it) result (flag)
        logical :: flag
        type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
        flag = vertex_particles_out_const_iterator_is_valid (it%obj, it%v_obj)
    end function hepmc_vertex_particle_out_iterator_is_valid

```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

```

<HepMC interface: interfaces>+≡
    interface

```

```

        type(c_ptr) function &
            vertex_particles_out_const_iterator_get (it_obj) bind(C)
        import
            type(c_ptr), value :: it_obj
        end function vertex_particles_out_const_iterator_get
    end interface

    <HepMC interface: public>+≡
        public :: hepmc_vertex_particle_out_iterator_get

    <HepMC interface: procedures>+≡
        function hepmc_vertex_particle_out_iterator_get (it) result (prt)
            type(hepmc_particle_t) :: prt
            type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
            prt%obj = vertex_particles_out_const_iterator_get (it%obj)
        end function hepmc_vertex_particle_out_iterator_get

```

### 17.6.8 GenEvent

The main object of HepMC is a GenEvent. The object is filled by GenVertex objects, which in turn contain GenParticle objects.

```

    <HepMC interface: public>+≡
        public :: hepmc_event_t

    <HepMC interface: types>+≡
        type :: hepmc_event_t
        private
            type(c_ptr) :: obj
        end type hepmc_event_t

```

Constructor. Arguments are process ID (integer) and event ID (integer).

The Fortran version has initializer form.

```

    <HepMC interface: interfaces>+≡
        interface
            type(c_ptr) function new_gen_event (proc_id, event_id) bind(C)
            import
                integer(c_int), value :: proc_id, event_id
            end function new_gen_event
        end interface

    <HepMC interface: public>+≡
        public :: hepmc_event_init

    <HepMC interface: procedures>+≡
        subroutine hepmc_event_init (evt, proc_id, event_id)
            type(hepmc_event_t), intent(out) :: evt
            integer, intent(in), optional :: proc_id, event_id
            integer(c_int) :: pid, eid
            pid = 0; if (present (proc_id)) pid = proc_id
            eid = 0; if (present (event_id)) eid = event_id
            evt%obj = new_gen_event (pid, eid)
        end subroutine hepmc_event_init

```

Destructor.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_delete (evt_obj) bind(C)
      import
      type(c_ptr), value :: evt_obj
    end subroutine gen_event_delete
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_final
<HepMC interface: procedures>+≡
  subroutine hepmc_event_final (evt)
    type(hepmc_event_t), intent(inout) :: evt
    call gen_event_delete (evt%obj)
  end subroutine hepmc_event_final

```

Screen output. Printing to file is possible in principle (using a C++ output channel), by allowing an argument. Printing to an open Fortran unit is obviously not possible.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_print (evt_obj) bind(C)
      import
      type(c_ptr), value :: evt_obj
    end subroutine gen_event_print
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_print
<HepMC interface: procedures>+≡
  subroutine hepmc_event_print (evt)
    type(hepmc_event_t), intent(in) :: evt
    call gen_event_print (evt%obj)
  end subroutine hepmc_event_print

```

Get the event number.

```

<HepMC interface: interfaces>+≡
  interface
    integer(c_int) function gen_event_event_number (evt_obj) bind(C)
      use iso_c_binding !NODEP!
      type(c_ptr), value :: evt_obj
    end function gen_event_event_number
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_get_event_index
<HepMC interface: procedures>+≡
  function hepmc_event_get_event_index (evt) result (i_proc)
    integer :: i_proc
    type(hepmc_event_t), intent(in) :: evt
    i_proc = gen_event_event_number (evt%obj)
  end function hepmc_event_get_event_index

```



Set the numeric signal process ID

```
<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_set_signal_process_id (evt_obj, proc_id) bind(C)
    import
      type(c_ptr), value :: evt_obj
      integer(c_int), value :: proc_id
    end subroutine gen_event_set_signal_process_id
  end interface

<HepMC interface: public>+≡
  public :: hepmc_event_set_process_id

<HepMC interface: procedures>+≡
  subroutine hepmc_event_set_process_id (evt, proc)
    type(hepmc_event_t), intent(in) :: evt
    integer, intent(in) :: proc
    integer(c_int) :: i_proc
    i_proc = proc
    call gen_event_set_signal_process_id (evt%obj, i_proc)
  end subroutine hepmc_event_set_process_id
```

Get the numeric signal process ID

```
<HepMC interface: interfaces>+≡
  interface
    integer(c_int) function gen_event_signal_process_id (evt_obj) bind(C)
    import
      type(c_ptr), value :: evt_obj
    end function gen_event_signal_process_id
  end interface

<HepMC interface: public>+≡
  public :: hepmc_event_get_process_id

<HepMC interface: procedures>+≡
  function hepmc_event_get_process_id (evt) result (i_proc)
    integer :: i_proc
    type(hepmc_event_t), intent(in) :: evt
    i_proc = gen_event_signal_process_id (evt%obj)
  end function hepmc_event_get_process_id
```

Set the event energy scale

```
<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_set_event_scale (evt_obj, scale) bind(C)
    import
      type(c_ptr), value :: evt_obj
      real(c_double), value :: scale
    end subroutine gen_event_set_event_scale
  end interface

<HepMC interface: public>+≡
  public :: hepmc_event_set_scale
```

```

<HepMC interface: procedures>+≡
subroutine hepmc_event_set_scale (evt, scale)
  type(hepmc_event_t), intent(in) :: evt
  real(default), intent(in) :: scale
  real(c_double) :: cscale
  cscale = scale
  call gen_event_set_event_scale (evt%obj, cscale)
end subroutine hepmc_event_set_scale

```

Get the event energy scale

```

<HepMC interface: interfaces>+≡
interface
  real(c_double) function gen_event_event_scale (evt_obj) bind(C)
    import
    type(c_ptr), value :: evt_obj
  end function gen_event_event_scale
end interface

```

```

<HepMC interface: public>+≡
public :: hepmc_event_get_scale

```

```

<HepMC interface: procedures>+≡
function hepmc_event_get_scale (evt) result (scale)
  real(default) :: scale
  type(hepmc_event_t), intent(in) :: evt
  scale = gen_event_event_scale (evt%obj)
end function hepmc_event_get_scale

```

Set the value of  $\alpha_{\text{QCD}}$ .

```

<HepMC interface: interfaces>+≡
interface
  subroutine gen_event_set_alpha_qcd (evt_obj, a) bind(C)
    import
    type(c_ptr), value :: evt_obj
    real(c_double), value :: a
  end subroutine gen_event_set_alpha_qcd
end interface

```

```

<HepMC interface: public>+≡
public :: hepmc_event_set_alpha_qcd

```

```

<HepMC interface: procedures>+≡
subroutine hepmc_event_set_alpha_qcd (evt, alpha)
  type(hepmc_event_t), intent(in) :: evt
  real(default), intent(in) :: alpha
  real(c_double) :: a
  a = alpha
  call gen_event_set_alpha_qcd (evt%obj, a)
end subroutine hepmc_event_set_alpha_qcd

```

Get the value of  $\alpha_{\text{QCD}}$ .

```

<HepMC interface: interfaces>+≡
interface
  real(c_double) function gen_event_alpha_qcd (evt_obj) bind(C)

```

```

        import
        type(c_ptr), value :: evt_obj
    end function gen_event_alpha_qcd
end interface

<HepMC interface: public>+≡
    public :: hepmc_event_get_alpha_qcd

<HepMC interface: procedures>+≡
    function hepmc_event_get_alpha_qcd (evt) result (alpha)
        real(default) :: alpha
        type(hepmc_event_t), intent(in) :: evt
        alpha = gen_event_alpha_qcd (evt%obj)
    end function hepmc_event_get_alpha_qcd

```

Set the value of  $\alpha_{\text{QED}}$ .

```

<HepMC interface: interfaces>+≡
    interface
        subroutine gen_event_set_alpha_qed (evt_obj, a) bind(C)
            import
            type(c_ptr), value :: evt_obj
            real(c_double), value :: a
        end subroutine gen_event_set_alpha_qed
    end interface

<HepMC interface: public>+≡
    public :: hepmc_event_set_alpha_qed

<HepMC interface: procedures>+≡
    subroutine hepmc_event_set_alpha_qed (evt, alpha)
        type(hepmc_event_t), intent(in) :: evt
        real(default), intent(in) :: alpha
        real(c_double) :: a
        a = alpha
        call gen_event_set_alpha_qed (evt%obj, a)
    end subroutine hepmc_event_set_alpha_qed

```

Get the value of  $\alpha_{\text{QED}}$ .

```

<HepMC interface: interfaces>+≡
    interface
        real(c_double) function gen_event_alpha_qed (evt_obj) bind(C)
            import
            type(c_ptr), value :: evt_obj
        end function gen_event_alpha_qed
    end interface

<HepMC interface: public>+≡
    public :: hepmc_event_get_alpha_qed

<HepMC interface: procedures>+≡
    function hepmc_event_get_alpha_qed (evt) result (alpha)
        real(default) :: alpha
        type(hepmc_event_t), intent(in) :: evt
        alpha = gen_event_alpha_qed (evt%obj)
    end function hepmc_event_get_alpha_qed

```

Clear a weight value to the end of the weight container.

```
<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_clear_weights (evt_obj) bind(C)
      use iso_c_binding !NODEP!
      type(c_ptr), value :: evt_obj
    end subroutine gen_event_clear_weights
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_clear_weights
<HepMC interface: procedures>+≡
  subroutine hepmc_event_clear_weights (evt)
    type(hepmc_event_t), intent(in) :: evt
    call gen_event_clear_weights (evt%obj)
  end subroutine hepmc_event_clear_weights
```

Add a weight value to the end of the weight container.

```
<HepMC interface: interfaces>+≡
  interface
    subroutine gen_event_add_weight (evt_obj, w) bind(C)
      use iso_c_binding !NODEP!
      type(c_ptr), value :: evt_obj
      real(c_double), value :: w
    end subroutine gen_event_add_weight
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_add_weight
<HepMC interface: procedures>+≡
  subroutine hepmc_event_add_weight (evt, weight)
    type(hepmc_event_t), intent(in) :: evt
    real(default), intent(in) :: weight
    real(c_double) :: w
    w = weight
    call gen_event_add_weight (evt%obj, w)
  end subroutine hepmc_event_add_weight
```

Get the size of the weight container (the number of valid elements).

```
<HepMC interface: interfaces>+≡
  interface
    integer(c_int) function gen_event_weights_size (evt_obj) bind(C)
      use iso_c_binding !NODEP!
      type(c_ptr), value :: evt_obj
    end function gen_event_weights_size
  end interface
<HepMC interface: public>+≡
  public :: hepmc_event_get_weights_size
<HepMC interface: procedures>+≡
  function hepmc_event_get_weights_size (evt) result (n)
    integer :: n
    type(hepmc_event_t), intent(in) :: evt
```

```

    n = gen_event_weights_size (evt%obj)
end function hepmc_event_get_weights_size

```

Get the value of the weight with index i. (Count from 1, while C counts from zero.)

```

<HepMC interface: interfaces>+≡
interface
    real(c_double) function gen_event_weight (evt_obj, i) bind(C)
    use iso_c_binding !NODEP!
    type(c_ptr), value :: evt_obj
    integer(c_int), value :: i
    end function gen_event_weight
end interface

<HepMC interface: public>+≡
public :: hepmc_event_get_weight

<HepMC interface: procedures>+≡
function hepmc_event_get_weight (evt, index) result (weight)
    real(default) :: weight
    type(hepmc_event_t), intent(in) :: evt
    integer, intent(in) :: index
    integer(c_int) :: i
    i = index - 1
    weight = gen_event_weight (evt%obj, i)
end function hepmc_event_get_weight

```

Add a vertex to the event container.

```

<HepMC interface: interfaces>+≡
interface
    subroutine gen_event_add_vertex (evt_obj, v_obj) bind(C)
    import
    type(c_ptr), value :: evt_obj
    type(c_ptr), value :: v_obj
    end subroutine gen_event_add_vertex
end interface

<HepMC interface: public>+≡
public :: hepmc_event_add_vertex

<HepMC interface: procedures>+≡
subroutine hepmc_event_add_vertex (evt, v)
    type(hepmc_event_t), intent(inout) :: evt
    type(hepmc_vertex_t), intent(in) :: v
    call gen_event_add_vertex (evt%obj, v%obj)
end subroutine hepmc_event_add_vertex

```

Mark a particular vertex as the signal process (hard interaction).

```

<HepMC interface: interfaces>+≡
interface
    subroutine gen_event_set_signal_process_vertex (evt_obj, v_obj) bind(C)
    import
    type(c_ptr), value :: evt_obj
    type(c_ptr), value :: v_obj

```

```

        end subroutine gen_event_set_signal_process_vertex
    end interface

    <HepMC interface: public>+≡
        public :: hepmc_event_set_signal_process_vertex

    <HepMC interface: procedures>+≡
        subroutine hepmc_event_set_signal_process_vertex (evt, v)
            type(hepmc_event_t), intent(inout) :: evt
            type(hepmc_vertex_t), intent(in) :: v
            call gen_event_set_signal_process_vertex (evt%obj, v%obj)
        end subroutine hepmc_event_set_signal_process_vertex

```

Set the beam particles explicitly.

```

    <HepMC interface: public>+≡
        public :: hepmc_event_set_beam_particles

    <HepMC interface: procedures>+≡
        subroutine hepmc_event_set_beam_particles (evt, prt1, prt2)
            type(hepmc_event_t), intent(inout) :: evt
            type(hepmc_particle_t), intent(in) :: prt1, prt2
            logical(c_bool) :: flag
            flag = gen_event_set_beam_particles (evt%obj, prt1%obj, prt2%obj)
        end subroutine hepmc_event_set_beam_particles

```

The C function returns a boolean which we do not use.

```

    <HepMC interface: interfaces>+≡
        interface
            logical(c_bool) function gen_event_set_beam_particles &
                (evt_obj, prt1_obj, prt2_obj) bind(C)
            import
            type(c_ptr), value :: evt_obj, prt1_obj, prt2_obj
            end function gen_event_set_beam_particles
        end interface

```

Set the cross section and error explicitly. Note that HepMC uses pb, while WHIZARD uses fb.

```

    <HepMC interface: public>+≡
        public :: hepmc_event_set_cross_section

    <HepMC interface: procedures>+≡
        subroutine hepmc_event_set_cross_section (evt, xsec, xsec_err)
            type(hepmc_event_t), intent(inout) :: evt
            real(default), intent(in) :: xsec, xsec_err
            call gen_event_set_cross_section &
                (evt%obj, &
                 real (xsec * 1e-3_default, c_double), &
                 real (xsec_err * 1e-3_default, c_double))
        end subroutine hepmc_event_set_cross_section

```

The C function returns a boolean which we do not use.

```

    <HepMC interface: interfaces>+≡
        interface

```

```

subroutine gen_event_set_cross_section (evt_obj, xs, xs_err) bind(C)
  import
  type(c_ptr), value :: evt_obj
  real(c_double), value :: xs, xs_err
end subroutine gen_event_set_cross_section
end interface

```

### 17.6.9 Event-particle iterator

This iterator iterates over all particles in an event. We store a pointer to the event in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

```

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_t

<HepMC interface: types>+≡
  type :: hepmc_event_particle_iterator_t
  private
  type(c_ptr) :: obj
  type(c_ptr) :: evt_obj
end type hepmc_event_particle_iterator_t

```

Constructor. The iterator is initialized at the first particle in the event.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function new_event_particle_const_iterator (evt_obj) bind(C)
    import
    type(c_ptr), value :: evt_obj
    end function new_event_particle_const_iterator
  end interface

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_init

<HepMC interface: procedures>+≡
  subroutine hepmc_event_particle_iterator_init (it, evt)
    type(hepmc_event_particle_iterator_t), intent(out) :: it
    type(hepmc_event_t), intent(in) :: evt
    it%obj = new_event_particle_const_iterator (evt%obj)
    it%evt_obj = evt%obj
  end subroutine hepmc_event_particle_iterator_init

```

Destructor. Necessary because the iterator is allocated on the heap.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine event_particle_const_iterator_delete (it_obj) bind(C)
    import
    type(c_ptr), value :: it_obj
    end subroutine event_particle_const_iterator_delete
  end interface

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_final

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_event_particle_iterator_final (it)
    type(hepmc_event_particle_iterator_t), intent(inout) :: it
    call event_particle_const_iterator_delete (it%obj)
  end subroutine hepmc_event_particle_iterator_final

```

Increment

```

<HepMC interface: interfaces>+≡
  interface
    subroutine event_particle_const_iterator_advance (it_obj) bind(C)
      import
      type(c_ptr), value :: it_obj
    end subroutine event_particle_const_iterator_advance
  end interface

```

```

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_advance

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_event_particle_iterator_advance (it)
    type(hepmc_event_particle_iterator_t), intent(inout) :: it
    call event_particle_const_iterator_advance (it%obj)
  end subroutine hepmc_event_particle_iterator_advance

```

Reset to the beginning

```

<HepMC interface: interfaces>+≡
  interface
    subroutine event_particle_const_iterator_reset (it_obj, evt_obj) bind(C)
      import
      type(c_ptr), value :: it_obj, evt_obj
    end subroutine event_particle_const_iterator_reset
  end interface

```

```

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_reset

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_event_particle_iterator_reset (it)
    type(hepmc_event_particle_iterator_t), intent(inout) :: it
    call event_particle_const_iterator_reset (it%obj, it%evt_obj)
  end subroutine hepmc_event_particle_iterator_reset

```

Test: return true as long as we are not past the end.

```

<HepMC interface: interfaces>+≡
  interface
    function event_particle_const_iterator_is_valid &
      (it_obj, evt_obj) result (flag) bind(C)
      import
      logical(c_bool) :: flag
      type(c_ptr), value :: it_obj, evt_obj
    end function event_particle_const_iterator_is_valid
  end interface

```

```

<HepMC interface: public>+≡
  public :: hepmc_event_particle_iterator_is_valid

```



```

<HepMC interface: procedures>+≡
function hepmc_event_particle_iterator_is_valid (it) result (flag)
  logical :: flag
  type(hepmc_event_particle_iterator_t), intent(in) :: it
  flag = event_particle_const_iterator_is_valid (it%obj, it%evt_obj)
end function hepmc_event_particle_iterator_is_valid

```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

```

<HepMC interface: interfaces>+≡
interface
  type(c_ptr) function event_particle_const_iterator_get (it_obj) bind(C)
  import
    type(c_ptr), value :: it_obj
  end function event_particle_const_iterator_get
end interface

```

```

<HepMC interface: public>+≡
public :: hepmc_event_particle_iterator_get

```

```

<HepMC interface: procedures>+≡
function hepmc_event_particle_iterator_get (it) result (prt)
  type(hepmc_particle_t) :: prt
  type(hepmc_event_particle_iterator_t), intent(in) :: it
  prt%obj = event_particle_const_iterator_get (it%obj)
end function hepmc_event_particle_iterator_get

```

### 17.6.10 I/O streams

There is a specific I/O stream type for handling the output of GenEvent objects (i.e., Monte Carlo event samples) to file. Opening the file is done by the constructor, closing by the destructor.

```

<HepMC interface: public>+≡
public :: hepmc_iostream_t

```

```

<HepMC interface: types>+≡
type :: hepmc_iostream_t
private
  type(c_ptr) :: obj
end type hepmc_iostream_t

```

Constructor for an output stream associated to a file.

```

<HepMC interface: interfaces>+≡
interface
  type(c_ptr) function new_io_gen_event_out (filename) bind(C)
  import
    character(c_char), dimension(*), intent(in) :: filename
  end function new_io_gen_event_out
end interface

```

```

<HepMC interface: public>+≡
public :: hepmc_iostream_open_out

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_iostream_open_out (iostream, filename)
    type(hepmc_iostream_t), intent(out) :: iostream
    type(string_t), intent(in) :: filename
    iostream%obj = new_io_gen_event_out (char (filename) // c_null_char)
  end subroutine hepmc_iostream_open_out

```

Constructor for an input stream associated to a file.

```

<HepMC interface: interfaces>+≡
  interface
    type(c_ptr) function new_io_gen_event_in (filename) bind(C)
    import
      character(c_char), dimension(*), intent(in) :: filename
    end function new_io_gen_event_in
  end interface

<HepMC interface: public>+≡
  public :: hepmc_iostream_open_in

<HepMC interface: procedures>+≡
  subroutine hepmc_iostream_open_in (iostream, filename)
    type(hepmc_iostream_t), intent(out) :: iostream
    type(string_t), intent(in) :: filename
    iostream%obj = new_io_gen_event_in (char (filename) // c_null_char)
  end subroutine hepmc_iostream_open_in

```

Destructor:

```

<HepMC interface: interfaces>+≡
  interface
    subroutine io_gen_event_delete (io_obj) bind(C)
    import
      type(c_ptr), value :: io_obj
    end subroutine io_gen_event_delete
  end interface

<HepMC interface: public>+≡
  public :: hepmc_iostream_close

<HepMC interface: procedures>+≡
  subroutine hepmc_iostream_close (iostream)
    type(hepmc_iostream_t), intent(inout) :: iostream
    call io_gen_event_delete (iostream%obj)
  end subroutine hepmc_iostream_close

```

Write a single event to the I/O stream.

```

<HepMC interface: interfaces>+≡
  interface
    subroutine io_gen_event_write_event (io_obj, evt_obj) bind(C)
    import
      type(c_ptr), value :: io_obj, evt_obj
    end subroutine io_gen_event_write_event
  end interface

<HepMC interface: public>+≡
  public :: hepmc_iostream_write_event

```

```

<HepMC interface: procedures>+≡
  subroutine hepmc_iostream_write_event (iostream, evt)
    type(hepmc_iostream_t), intent(inout) :: iostream
    type(hepmc_event_t), intent(in) :: evt
    call io_gen_event_write_event (iostream%obj, evt%obj)
  end subroutine hepmc_iostream_write_event

```

Read a single event from the I/O stream. Return true if successful.

```

<HepMC interface: interfaces>+≡
  interface
    logical(c_bool) function io_gen_event_read_event (io_obj, evt_obj) bind(C)
      import
      type(c_ptr), value :: io_obj, evt_obj
    end function io_gen_event_read_event
  end interface

<HepMC interface: public>+≡
  public :: hepmc_iostream_read_event

<HepMC interface: procedures>+≡
  subroutine hepmc_iostream_read_event (iostream, evt, ok)
    type(hepmc_iostream_t), intent(inout) :: iostream
    type(hepmc_event_t), intent(in) :: evt
    logical, intent(out) :: ok
    ok = io_gen_event_read_event (iostream%obj, evt%obj)
  end subroutine hepmc_iostream_read_event

```

### 17.6.11 Test

This test example is an abridged version from the build-from-scratch example in the HepMC distribution. We create two vertices for  $p \rightarrow q$  PDF splitting, then a vertex for a  $qq \rightarrow W^- g$  hard-interaction process, and finally a vertex for  $W^- \rightarrow qq$  decay. The setup is for LHC kinematics.

Extending the original example, we set color flow for the incoming quarks and polarization for the outgoing photon. For the latter, we have to define a particle-data object for the photon, so a flavor object can be correctly initialized.

```

<CCC HepMC interface: public>≡
  public :: hepmc_test

<CCC HepMC interface: procedures>≡
  subroutine hepmc_test
    type(hepmc_event_t) :: evt
    type(hepmc_vertex_t) :: v1, v2, v3, v4
    type(hepmc_particle_t) :: prt1, prt2, prt3, prt4, prt5, prt6, prt7, prt8
    type(hepmc_iostream_t) :: iostream
    type(flavor_t) :: flv
    type(color_t) :: col
    type(polarization_t) :: pol
    type(particle_data_t), target :: photon_data

    ! Initialize a photon flavor object and some polarization
    call particle_data_init (photon_data, var_str ("PHOTON"), 22)
    call particle_data_set (photon_data, spin_type=VECTOR)
  end subroutine hepmc_test

```

```

call particle_data_freeze (photon_data)
call flavor_init (flv, photon_data)
call polarization_init_angles &
    (pol, flv, 0.6_default, 1._default, 0.5_default)

! Event initialization
call hepmc_event_init (evt, 20, 1)

! $p\to q$ splittings
call hepmc_vertex_init (v1)
call hepmc_event_add_vertex (evt, v1)
call hepmc_vertex_init (v2)
call hepmc_event_add_vertex (evt, v2)
call particle_init (prt1, &
    0._default, 0._default, 7000._default, 7000._default, &
    2212, 3)
call hepmc_vertex_add_particle_in (v1, prt1)
call particle_init (prt2, &
    0._default, 0._default, -7000._default, 7000._default, &
    2212, 3)
call hepmc_vertex_add_particle_in (v2, prt2)
call particle_init (prt3, &
    .750_default, -1.569_default, 32.191_default, 32.238_default, &
    1, 3)
call color_init_from_array (col, (/501/))
call hepmc_particle_set_color (prt3, col)
call hepmc_vertex_add_particle_out (v1, prt3)
call particle_init (prt4, &
    -3.047_default, -19._default, -54.629_default, 57.920_default, &
    -2, 3)
call color_init_from_array (col, (/ -501 /))
call hepmc_particle_set_color (prt4, col)
call hepmc_vertex_add_particle_out (v2, prt4)

! Hard interaction
call hepmc_vertex_init (v3)
call hepmc_event_add_vertex (evt, v3)
call hepmc_vertex_add_particle_in (v3, prt3)
call hepmc_vertex_add_particle_in (v3, prt4)
call particle_init (prt6, &
    -3.813_default, 0.113_default, -1.833_default, 4.233_default, &
    22, 1)
call hepmc_particle_set_polarization (prt6, pol)
call hepmc_vertex_add_particle_out (v3, prt6)
call particle_init (prt5, &
    1.517_default, -20.68_default, -20.605_default, 85.925_default, &
    -24, 3)
call hepmc_vertex_add_particle_out (v3, prt5)
call hepmc_event_set_signal_process_vertex (evt, v3)

! $W^-$ decay
call vertex_init_pos (v4, &
    0.12_default, -0.3_default, 0.05_default, 0.004_default)
call hepmc_event_add_vertex (evt, v4)

```

```

call hepmc_vertex_add_particle_in (v4, prt5)
call particle_init (prt7, &
  -2.445_default, 28.816_default, 6.082_default, 29.552_default, &
  1, 1)
call hepmc_vertex_add_particle_out (v4, prt7)
call particle_init (prt8, &
  3.962_default, -49.498_default, -26.687_default, 56.373_default, &
  -2, 1)
call hepmc_vertex_add_particle_out (v4, prt8)

! Event output
call hepmc_event_print (evt)
print *, "Writing to file 'hepmc_test.hepmc.dat'"
call hepmc_iostream_open_out (iostream, var_str ("hepmc_test.hepmc.dat"))
call hepmc_iostream_write_event (iostream, evt)
call hepmc_iostream_close (iostream)
print *, "Write completed"

! Wrapup
call polarization_final (pol)
call hepmc_event_final (evt)

contains

subroutine vertex_init_pos (v, x, y, z, t)
  type(hepmc_vertex_t), intent(out) :: v
  real(default), intent(in) :: x, y, z, t
  type(vector4_t) :: xx
  xx = vector4_moving (t, vector3_moving ((/x, y, z/)))
  call hepmc_vertex_init (v, xx)
end subroutine vertex_init_pos

subroutine particle_init (prt, px, py, pz, E, pdg, status)
  type(hepmc_particle_t), intent(out) :: prt
  real(default), intent(in) :: px, py, pz, E
  integer, intent(in) :: pdg, status
  type(vector4_t) :: p
  p = vector4_moving (E, vector3_moving ((/px, py, pz/)))
  call hepmc_particle_init (prt, p, pdg, status)
end subroutine particle_init

end subroutine hepmc_test

```

## 17.7 HepMC Output

The HepMC event record is standardized. It is an ASCII format. We try our best at using it for both input and output.

```

(eio_hepmc.f90)≡
  <File header>

```

```

module eio_hepmc

```

```

    use kinds !NODEP!
    <Use file utils>
    <Use strings>
    use diagnostics !NODEP!
    use unit_tests

    use lorentz !NODEP!
    use particles
    use subevents
    use beams
    use processes
    use events
    use eio_data
    use eio_base
    use hepmc_interface

    <Standard module head>

    <EIO HepMC: public>

    <EIO HepMC: types>

contains

    <EIO HepMC: procedures>

    <EIO HepMC: tests>

end module eio_hepmc

```

### 17.7.1 Type

```

<EIO HepMC: public>≡
    public :: eio_hepmc_t

<EIO HepMC: types>≡
    type, extends (eio_t) :: eio_hepmc_t
        type(string_t) :: filename
        logical :: writing = .false.
        logical :: keep_beams = .false.
        type(hepmc_iostream_t) :: iostream
    contains
        <EIO HepMC: eio hepmc: TBP>
    end type eio_hepmc_t

```

### 17.7.2 Specific Methods

Set parameters that are specifically used with HepMC.

```

<EIO HepMC: eio hepmc: TBP>≡
    procedure :: set_parameters => eio_hepmc_set_parameters

```

```

(EIO HepMC: procedures)≡
  subroutine eio_hepmc_set_parameters (eio, keep_beams)
    class(eio_hepmc_t), intent(inout) :: eio
    logical, intent(in), optional :: keep_beams
    if (present (keep_beams)) eio%keep_beams = keep_beams
  end subroutine eio_hepmc_set_parameters

```

### 17.7.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

```

(EIO HepMC: eio hepmc: TBP)+≡
  procedure :: write => eio_hepmc_write

(EIO HepMC: procedures)+≡
  subroutine eio_hepmc_write (object, unit)
    class(eio_hepmc_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "HepMC event stream:"
    if (object%writing) then
      write (u, "(3x,A,A)") "Writing to file   = ", char (object%filename)
    else
      write (u, "(3x,A)") "[closed]"
    end if
    write (u, "(3x,A,L1)") "Keep beams       = ", object%keep_beams
  end subroutine eio_hepmc_write

```

Finalizer: close any open file.

```

(EIO HepMC: eio hepmc: TBP)+≡
  procedure :: final => eio_hepmc_final

(EIO HepMC: procedures)+≡
  subroutine eio_hepmc_final (object)
    class(eio_hepmc_t), intent(inout) :: object
    if (object%writing) then
      write (msg_buffer, "(A,A,A)") "Events: closing HepMC file '", &
        char (object%filename), "'"
      call msg_message ()
      call hepmc_iostream_close (object%iostream)
      object%writing = .false.
    end if
  end subroutine eio_hepmc_final

```

Initialize event writing.

```

(EIO HepMC: eio hepmc: TBP)+≡
  procedure :: init_out => eio_hepmc_init_out

```

*<EIO HepMC: procedures>+≡*

```

subroutine eio_hepmc_init_out (eio, sample, process_ptr, data, success)
  class(eio_hepmc_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  integer :: i
  if (.not. present (data)) &
    call msg_bug ("HepMC initialization: missing data")
  if (data%n_beam /= 2) &
    call msg_fatal ("HepMC: defined for scattering processes only")
  eio%filename = sample // ".hepmc"
  write (msg_buffer, "(A,A,A)") "Events: writing to HepMC file '", &
    char (eio%filename), "'"
  call msg_message ()
  eio%writing = .true.
  call hepmc_iostream_open_out (eio%iostream, eio%filename)
  if (present (success)) success = .true.
end subroutine eio_hepmc_init_out

```

Initialize event reading.

*<EIO HepMC: eio hepmc: TBP>+≡*

```

procedure :: init_in => eio_hepmc_init_in

```

*<EIO HepMC: procedures>+≡*

```

subroutine eio_hepmc_init_in (eio, sample, process_ptr, data, success)
  class(eio_hepmc_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  call msg_bug ("HepMC: event input not supported")
end subroutine eio_hepmc_init_in

```

Switch from input to output: reopen the file for reading.

*<EIO HepMC: eio hepmc: TBP>+≡*

```

procedure :: switch_inout => eio_hepmc_switch_inout

```

*<EIO HepMC: procedures>+≡*

```

subroutine eio_hepmc_switch_inout (eio, success)
  class(eio_hepmc_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  call msg_bug ("HepMC: in-out switch not supported")
end subroutine eio_hepmc_switch_inout

```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

*<EIO HepMC: eio hepmc: TBP>+≡*

```

procedure :: output => eio_hepmc_output

```

*<EIO HepMC: procedures>+≡*

```

subroutine eio_hepmc_output (eio, event, i_prc)

```



```

class(eio_hepmc_t), intent(inout) :: eio
type(event_t), intent(in), target :: event
integer, intent(in) :: i_prc
!   real(default) :: weight, sqme_evt, sqme_prc
type(hepmc_event_t) :: hepmc_event
type(particle_set_t), pointer :: pset_ptr
if (eio%writing) then
    pset_ptr => event%get_particle_set_ptr ()
    call hepmc_event_init (hepmc_event, proc_id = i_prc)
    call hepmc_event_from_particle_set (hepmc_event, pset_ptr)
    call hepmc_iostream_write_event (eio%iostream, hepmc_event)
    call hepmc_event_final (hepmc_event)
else
    call eio%write ()
    call msg_fatal ("HepMC file is not open for writing")
end if
end subroutine eio_hepmc_output

```

Input an event.

```

<EIO HepMC: eio hepmc: TBP>+≡
    procedure :: input_i_prc => eio_hepmc_input_i_prc
    procedure :: input_event => eio_hepmc_input_event

<EIO HepMC: procedures>+≡
    subroutine eio_hepmc_input_i_prc (eio, i_prc, iostat)
        class(eio_hepmc_t), intent(inout) :: eio
        integer, intent(out) :: i_prc
        integer, intent(out) :: iostat
        call msg_bug ("HepMC: event input not supported")
    end subroutine eio_hepmc_input_i_prc

    subroutine eio_hepmc_input_event (eio, event, iostat)
        class(eio_hepmc_t), intent(inout) :: eio
        type(event_t), intent(inout), target :: event
        integer, intent(out) :: iostat
        call msg_bug ("HepMC: event input not supported")
    end subroutine eio_hepmc_input_event

```

#### 17.7.4 Particle Set Transfer

The master output function fills a HepMC GenEvent object that is already initialized, but has no vertices in it.

We first set up the vertex lists and enter the vertices into the HepMC event. Then, we assign first all incoming particles and then all outgoing particles to their associated vertices. Particles which have neither parent nor children entries (this should not happen) are dropped.

Finally, we insert the beam particles. If there are none, use the incoming particles instead.

```

<EIO HepMC: procedures>+≡
    subroutine hepmc_event_from_particle_set (evt, particle_set)
        type(hepmc_event_t), intent(inout) :: evt

```

```

type(particle_set_t), intent(in) :: particle_set
type(hepmc_vertex_t), dimension(:), allocatable :: v
type(hepmc_particle_t), dimension(:), allocatable :: hpvt
type(hepmc_particle_t), dimension(2) :: hbeam
logical, dimension(:), allocatable :: is_beam
integer, dimension(:), allocatable :: v_from, v_to
integer :: n_vertices, n_tot, i
n_tot = particle_set%n_tot
allocate (v_from (n_tot), v_to (n_tot))
call particle_set_assign_vertices (particle_set, v_from, v_to, n_vertices)
allocate (v (n_vertices))
do i = 1, n_vertices
    call hepmc_vertex_init (v(i))
    call hepmc_event_add_vertex (evt, v(i))
end do
allocate (hpvt (n_tot))
do i = 1, n_tot
    if (v_to(i) /= 0 .or. v_from(i) /= 0) then
        call particle_to_hepmc (particle_set%prt(i), hpvt(i))
    end if
end do
allocate (is_beam (n_tot))
is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_BEAM
if (.not. any (is_beam)) then
    is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_INCOMING
end if
if (count (is_beam) == 2) then
    hbeam = pack (hpvt, is_beam)
    call hepmc_event_set_beam_particles (evt, hbeam(1), hbeam(2))
end if
do i = 1, n_tot
    if (v_to(i) /= 0) then
        call hepmc_vertex_add_particle_in (v(v_to(i)), hpvt(i))
    end if
end do
do i = 1, n_tot
    if (v_from(i) /= 0) then
        call hepmc_vertex_add_particle_out (v(v_from(i)), hpvt(i))
    end if
end do
end subroutine hepmc_event_from_particle_set

```

Transform a particle into a `hepmc_particle` object, including color and polarization. The HepMC status is equivalent to the HEPEVT status, in particular: 0 = null entry, 1 = physical particle, 2 = decayed/fragmented SM hadron, tau or muon, 3 = other unphysical particle entry, 4 = incoming particles, 11 = intermediate resonance such as squarks. The use of 11 for intermediate resonances is as done by HERWIG, see <http://herwig.hepforge.org/trac/wiki/FaQs>.

*(EIO HepMC: procedures)+≡*

```

subroutine particle_to_hepmc (prt, hpvt)
    type(particle_t), intent(in) :: prt
    type(hepmc_particle_t), intent(out) :: hpvt
    integer :: hepmc_status

```

```

select case (particle_get_status (prt))
case (PRT_UNDEFINED)
    hepmc_status = 0
case (PRT_OUTGOING)
    hepmc_status = 1
case (PRT_BEAM)
    hepmc_status = 4
case (PRT_RESONANT)
    if(abs(particle_get_pdg(prt)) == 13 .or. &
       abs(particle_get_pdg(prt)) == 15) then
        hepmc_status = 2
    else
        hepmc_status = 11
    end if
case default
    hepmc_status = 3
end select
call hepmc_particle_init (hpvt, &
    particle_get_momentum (prt), &
    particle_get_pdg (prt), &
    hepmc_status)
call hepmc_particle_set_color (hpvt, &
    particle_get_color (prt))
select case (particle_get_polarization_status (prt))
case (PRT_DEFINITE_HELICITY)
    call hepmc_particle_set_polarization (hpvt, &
        particle_get_helicity (prt))
case (PRT_GENERIC_POLARIZATION)
    call hepmc_particle_set_polarization (hpvt, &
        particle_get_polarization (prt))
end select
end subroutine particle_to_hepmc

```

### 17.7.5 Unit tests

```

<EIO HepMC: public>+≡
public :: eio_hepmc_test

<EIO HepMC: tests>≡
subroutine eio_hepmc_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
<EIO HepMC: execute tests>
end subroutine eio_hepmc_test

```

### Test I/O methods

We test the implementation of all I/O methods.

```

<EIO HepMC: execute tests>≡
call test (eio_hepmc_1, "eio_hepmc_1", &
    "read and write event contents", &
    u, results)

```

*<EIO HepMC: tests>+≡*

```

subroutine eio_hepmc_1 (u)
  integer, intent(in) :: u
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: eio_hepmc_1"
  write (u, "(A)")  "* Purpose: generate an event and write weight to file"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  process_ptr%ptr => process
  allocate (process_instance)
  call prepare_test_process (process, process_instance)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%connect (process_instance)

  call data%init (1)
  data%n_beam = 2
  data%pdg_beam = 25
  data%energy_beam = 500
  data%cross_section(1) = 100
  data%error(1) = 1

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_hepmc_1"

  allocate (eio_hepmc_t :: eio)

  call eio%init_out (sample, [process_ptr], data)
  call event%generate (1, [0._default, 0._default])

  call eio%output (event, i_prc = 42)
  call eio%write (u)
  call eio%final ()

  write (u, "(A)")
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")

```

```

open (u_file, file = "eio_hepmc_1.hepmc", &
      action = "read", status = "old")
do
  read (u_file, "(A)", iostat = iostat)  buffer
  if (iostat /= 0) exit
  if (trim (buffer) == "") cycle
  if (buffer(1:14) == "HepMC::Version") cycle
  write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_hepmc_t :: eio)

select type (eio)
type is (eio_hepmc_t)
  call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_hepmc_1"

end subroutine eio_hepmc_1

```

## 17.8 Event Weight Output

This is an output-only format. For each event, we print the indices that identify process, process part (MCI group), and term. As numerical information we print the squared matrix element (trace) and the event weight.

```

(eio_weights.f90)≡
  <File header>

  module eio_weights

    use kinds !NODEP!
    <Use file utils>

```

```

<Use strings>
  use diagnostics !NODEP!
  use unit_tests

  use lorentz !NODEP!
  use particles
  use beams
  use processes
  use events
  use eio_data
  use eio_base

<Standard module head>

<EIO weights: public>

<EIO weights: types>

contains

<EIO weights: procedures>

<EIO weights: tests>

end module eio_weights

```

### 17.8.1 Type

```

<EIO weights: public>≡
  public :: eio_weights_t

<EIO weights: types>≡
  type, extends (eio_t) :: eio_weights_t
    type(string_t) :: filename
    logical :: writing = .false.
    integer :: unit = 0
  contains
    <EIO weights: eio weights: TBP>
  end type eio_weights_t

```

Output. This is not the actual event format, but a readable account of the current object status.

```

<EIO weights: eio weights: TBP>≡
  procedure :: write => eio_weights_write

<EIO weights: procedures>≡
  subroutine eio_weights_write (object, unit)
    class(eio_weights_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, "(1x,A)") "Weight stream:"
    if (object%writing) then
      write (u, "(3x,A,A)") "Writing to file  = ", char (object%filename)
    end if
  end subroutine eio_weights_write

```

```

else
    write (u, "(3x,A)") "[closed]"
end if
end subroutine eio_weights_write

```

Finalizer: close any open file.

```

(EIO weights: eio weights: TBP)+≡
    procedure :: final => eio_weights_final

(EIO weights: procedures)+≡
    subroutine eio_weights_final (object)
        class(eio_weights_t), intent(inout) :: object
        if (object%writing) then
            write (msg_buffer, "(A,A,A)") "Events: closing weight stream file '", &
                char (object%filename), "'"
            call msg_message ()
            close (object%unit)
            object%writing = .false.
        end if
    end subroutine eio_weights_final

```

Initialize event writing.

```

(EIO weights: eio weights: TBP)+≡
    procedure :: init_out => eio_weights_init_out

(EIO weights: procedures)+≡
    subroutine eio_weights_init_out &
        (eio, sample, process_ptr, data, success)
        class(eio_weights_t), intent(inout) :: eio
        type(string_t), intent(in) :: sample
        type(process_ptr_t), dimension(:), intent(in) :: process_ptr
        type(event_sample_data_t), intent(in), optional :: data
        logical, intent(out), optional :: success
        eio%filename = sample // ".weights.dat"
        eio%unit = free_unit ()
        write (msg_buffer, "(A,A,A)") "Events: writing to weight stream file '", &
            char (eio%filename), "'"
        call msg_message ()
        eio%writing = .true.
        open (eio%unit, file = char (eio%filename), &
            action = "write", status = "replace")
        if (present (success)) success = .true.
    end subroutine eio_weights_init_out

```

Initialize event reading.

```

(EIO weights: eio weights: TBP)+≡
    procedure :: init_in => eio_weights_init_in

(EIO weights: procedures)+≡
    subroutine eio_weights_init_in (eio, sample, process_ptr, data, success)
        class(eio_weights_t), intent(inout) :: eio
        type(string_t), intent(in) :: sample
        type(process_ptr_t), dimension(:), intent(in) :: process_ptr
        type(event_sample_data_t), intent(in), optional :: data

```

```

        logical, intent(out), optional :: success
        call msg_bug ("Weight stream: event input not supported")
    end subroutine eio_weights_init_in

```

Switch from input to output: reopen the file for reading.

```

<EIO weights: eio weights: TBP>+≡
    procedure :: switch_inout => eio_weights_switch_inout

<EIO weights: procedures>+≡
    subroutine eio_weights_switch_inout (eio, success)
        class(eio_weights_t), intent(inout) :: eio
        logical, intent(out), optional :: success
        call msg_bug ("Weight stream: in-out switch not supported")
    end subroutine eio_weights_switch_inout

```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

```

<EIO weights: eio weights: TBP>+≡
    procedure :: output => eio_weights_output

<EIO weights: procedures>+≡
    subroutine eio_weights_output (eio, event, i_prc)
        class(eio_weights_t), intent(inout) :: eio
        type(event_t), intent(in), target :: event
        integer, intent(in) :: i_prc
        integer :: i_mci, i_term
        real(default) :: weight, sqme_evt, sqme_prc
        if (eio%writing) then
            i_mci = event%get_i_mci ()
            i_term = event%get_i_term ()
            weight = event%get_weight ()
            sqme_evt = event%get_sqme ()
            sqme_prc = event%get_sqme_process ()
1      format (I0,1x,I0,1x,I0,3(1x,ES19.12))
            write (eio%unit, 1) i_prc, i_mci, i_term, weight, sqme_evt, sqme_prc
        else
            call eio%write ()
            call msg_fatal ("Weight stream file is not open for writing")
        end if
    end subroutine eio_weights_output

```

Input an event.

```

<EIO weights: eio weights: TBP>+≡
    procedure :: input_i_prc => eio_weights_input_i_prc
    procedure :: input_event => eio_weights_input_event

<EIO weights: procedures>+≡
    subroutine eio_weights_input_i_prc (eio, i_prc, iostat)
        class(eio_weights_t), intent(inout) :: eio
        integer, intent(out) :: i_prc
        integer, intent(out) :: iostat
        call msg_bug ("Weight stream: event input not supported")
    end subroutine eio_weights_input_i_prc

```



```

subroutine eio_weights_input_event (eio, event, iostat)
  class(eio_weights_t), intent(inout) :: eio
  type(event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  call msg_bug ("Weight stream: event input not supported")
end subroutine eio_weights_input_event

```

## 17.8.2 Unit tests

```

<EIO weights: public>+≡
  public :: eio_weights_test

<EIO weights: tests>≡
  subroutine eio_weights_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <EIO weights: execute tests>
  end subroutine eio_weights_test

```

### Test I/O methods

We test the implementation of all I/O methods.

```

<EIO weights: execute tests>≡
  call test (eio_weights_1, "eio_weights_1", &
    "read and write event contents", &
    u, results)

<EIO weights: tests>+≡
  subroutine eio_weights_1 (u)
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_ptr_t) :: process_ptr
    type(process_instance_t), allocatable, target :: process_instance
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_weights_1"
    write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    allocate (process)
    process_ptr%ptr => process
    allocate (process_instance)
    call prepare_test_process (process, process_instance)
    call process_instance%setup_event_data ()

```

```

allocate (event)
call event%connect (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_weights_1"

allocate (eio_weights_t :: eio)

call eio%init_out (sample, [process_ptr])
call event%generate (1, [0._default, 0._default])

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents: &
    &(i_prc, i_mci, i_term, weight, sqme(evt), sqme(prc))"
write (u, "(A)")

open (u_file, file = "eio_weights_1.weights.dat", &
    action = "read", status = "old")
read (u_file, "(A)") buffer
write (u, "(A)") trim (buffer)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_weights_1"

end subroutine eio_weights_1

```

## Chapter 18

# The SUSY Les Houches Accord

The SUSY Les Houches Accord defines a standard interfaces for storing the physics data of SUSY models. Here, we provide the means for reading, storing, and writing such data.

`<slha_interface.f90>`≡  
*<File header>*

```
module slha_interface
```

```
<Use kinds>
```

```
<Use strings>
```

```
  use limits, only: EOF, VERSION_STRING !NODEP!
```

```
  use constants !NODEP!
```

```
<Use file utils>
```

```
  use diagnostics !NODEP!
```

```
  use os_interface
```

```
  use ifiles
```

```
  use lexers
```

```
  use syntax_rules
```

```
  use parser
```

```
  use variables
```

```
  use expressions
```

```
  use models
```

```
<Standard module head>
```

```
<SLHA: public>
```

```
<SLHA: parameters>
```

```
<SLHA: variables>
```

```
  save
```

```
contains
```

*<SLHA: procedures>*

end module slha\_interface

### 18.0.3 Preprocessor

SLHA is a mixed-format standard. It should be read in assuming free format (but line-oriented), but it has some fixed-format elements.

To overcome this difficulty, we implement a preprocessing step which transforms the SLHA into a format that can be swallowed by our generic free-format lexer and parser. Each line with a blank first character is assumed to be a data line. We prepend a 'DATA' keyword to these lines. Furthermore, to enforce line-orientation, each line is appended a '\$' key which is recognized by the parser. To do this properly, we first remove trailing comments, and skip lines consisting only of comments.

The preprocessor reads from a stream and puts out an `ifile`. Blocks that are not recognized are skipped. For some blocks, data items are quoted, so they can be read as strings if necessary.

*<SLHA: parameters>*≡

```
integer, parameter :: MODE_SKIP = 0, MODE_DATA = 1, MODE_INFO = 2
```

*<SLHA: procedures>*≡

```
subroutine slha_preprocess (stream, ifile)
  type(stream_t), intent(inout), target :: stream
  type(ifile_t), intent(out) :: ifile
  type(string_t) :: buffer, line, item
  integer :: iostat
  integer :: mode
  mode = MODE
  SCAN_FILE: do
    call stream_get_record (stream, buffer, iostat)
    select case (iostat)
    case (0)
      call split (buffer, line, "#")
      if (len_trim (line) == 0) cycle SCAN_FILE
      select case (char (extract (line, 1, 1)))
      case ("B", "b")
        mode = check_block_handling (line)
        call ifile_append (ifile, line // "$")
      case ("D", "d")
        mode = MODE_DATA
        call ifile_append (ifile, line // "$")
      case (" ")
        select case (mode)
        case (MODE_DATA)
          call ifile_append (ifile, "DATA" // line // "$")
        case (MODE_INFO)
          line = adjustl (line)
          call split (line, item, " ")
          call ifile_append (ifile, "INFO" // " " // item // " " &
            // ' ' // trim (adjustl (line)) // ' ' '$')
        end select
    end select
```

```

        case default
            call msg_message (char (line))
            call msg_fatal ("SLHA: Incomprehensible line")
        end select
    case (EOF)
        exit SCAN_FILE
    case default
        call msg_fatal ("SLHA: I/O error occured while reading SLHA input")
    end select
end do SCAN_FILE
end subroutine slha_preprocess

```

Return the mode that we should treat this block with. We need to recognize only those blocks that we actually use.

```

⟨SLHA: procedures⟩+≡
function check_block_handling (line) result (mode)
    integer :: mode
    type(string_t), intent(in) :: line
    type(string_t) :: buffer, key, block_name
    buffer = trim (line)
    call split (buffer, key, " ")
    buffer = adjustl (buffer)
    call split (buffer, block_name, " ")
    block_name = trim (adjustl (upper_case (block_name)))
    select case (char (block_name))
    case ("MODSEL", "MINPAR", "SMINPUTS")
        mode = MODE_DATA
    case ("MASS")
        mode = MODE_DATA
    case ("NMIX", "UMIX", "VMIX", "STOPMIX", "SBOTMIX", "STAUMIX")
        mode = MODE_DATA
    case ("NMHMX", "NMAMIX", "NMNMIX", "NMSSMRUN")
        mode = MODE_DATA
    case ("ALPHA", "HMIX")
        mode = MODE_DATA
    case ("AU", "AD", "AE")
        mode = MODE_DATA
    case ("SPINFO", "DCINFO")
        mode = MODE_INFO
    case default
        mode = MODE_SKIP
    end select
end function check_block_handling

```

#### 18.0.4 Lexer and syntax

```

⟨SLHA: variables⟩≡
type(syntax_t), target :: syntax_slha

```

```

⟨SLHA: public⟩≡
public :: syntax_slha_init

```

```

<SLHA: procedures> +=
  subroutine syntax_slha_init ()
    type(ifile_t) :: ifile
    call define_slha_syntax (ifile)
    call syntax_init (syntax_slha, ifile)
    call ifile_final (ifile)
  end subroutine syntax_slha_init

<SLHA: public> +=
  public :: syntax_slha_final

<SLHA: procedures> +=
  subroutine syntax_slha_final ()
    call syntax_final (syntax_slha)
  end subroutine syntax_slha_final

<SLHA: public> +=
  public :: syntax_slha_write

<SLHA: procedures> +=
  subroutine syntax_slha_write (unit)
    integer, intent(in), optional :: unit
    call syntax_write (syntax_slha, unit)
  end subroutine syntax_slha_write

<SLHA: procedures> +=
  subroutine define_slha_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ slha = chunk*")
    call ifile_append (ifile, "ALT chunk = block_def | decay_def")
    call ifile_append (ifile, "SEQ block_def = " &
      // "BLOCK block_spec '$' block_line*")
    call ifile_append (ifile, "KEY BLOCK")
    call ifile_append (ifile, "SEQ block_spec = block_name qvalue?")
    call ifile_append (ifile, "IDE block_name")
    call ifile_append (ifile, "SEQ qvalue = qname '=' real")
    call ifile_append (ifile, "IDE qname")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "REA real")
    call ifile_append (ifile, "KEY '$'")
    call ifile_append (ifile, "ALT block_line = block_data | block_info")
    call ifile_append (ifile, "SEQ block_data = DATA data_line '$'")
    call ifile_append (ifile, "KEY DATA")
    call ifile_append (ifile, "SEQ data_line = data_item+")
    call ifile_append (ifile, "ALT data_item = signed_number | number")
    call ifile_append (ifile, "SEQ signed_number = sign number")
    call ifile_append (ifile, "ALT sign = '+' | '-'")
    call ifile_append (ifile, "ALT number = integer | real")
    call ifile_append (ifile, "INT integer")
    call ifile_append (ifile, "KEY '-'")
    call ifile_append (ifile, "KEY '+'")
    call ifile_append (ifile, "SEQ block_info = INFO info_line '$'")
    call ifile_append (ifile, "KEY INFO")
    call ifile_append (ifile, "SEQ info_line = integer string_literal")
  end subroutine define_slha_syntax

```

```

call ifile_append (ifile, "QUO string_literal = '""'...'""')
call ifile_append (ifile, "SEQ decay_def = " &
// "DECAY decay_spec '$' decay_data*")
call ifile_append (ifile, "KEY DECAY")
call ifile_append (ifile, "SEQ decay_spec = pdg_code data_item")
call ifile_append (ifile, "ALT pdg_code = signed_integer | integer")
call ifile_append (ifile, "SEQ signed_integer = sign integer")
call ifile_append (ifile, "SEQ decay_data = DATA decay_line '$'")
call ifile_append (ifile, "SEQ decay_line = data_item integer pdg_code+")
end subroutine define_slha_syntax

```

The SLHA specification allows for string data items in certain places. Currently, we do not interpret them, but the strings, which are not quoted, must be parsed somehow. The hack for this problem is to allow essentially all characters as special characters, so the string can be read before it is discarded.

```

<SLHA: public>+=
public :: lexer_init_slha

<SLHA: procedures>+=
subroutine lexer_init_slha (lexer)
type(lexer_t), intent(out) :: lexer
call lexer_init (lexer, &
comment_chars = "#", &
quote_chars = "'", &
quote_match = "'", &
single_chars = "+-=$", &
special_class = (/ "" /), &
keyword_list = syntax_get_keyword_list_ptr (syntax_slha), &
upper_case_keywords = .true.) ! $
end subroutine lexer_init_slha

```

## 18.0.5 Interpreter

### Find blocks

From the parse tree, find the node that represents a particular block. If `required` is true, issue an error if not found. Since `block_name` is always invoked with capital letters, we have to capitalize `pn_block_name`.

```

<SLHA: procedures>+=
function slha_get_block_ptr &
(parse_tree, block_name, required) result (pn_block)
type(parse_node_t), pointer :: pn_block
type(parse_tree_t), intent(in) :: parse_tree
type(string_t), intent(in) :: block_name
logical, intent(in) :: required
type(parse_node_t), pointer :: pn_root, pn_block_spec, pn_block_name
pn_root => parse_tree_get_root_ptr (parse_tree)
pn_block => parse_node_get_sub_ptr (pn_root)
do while (associated (pn_block))
select case (char (parse_node_get_rule_key (pn_block)))
case ("block_def")
pn_block_spec => parse_node_get_sub_ptr (pn_block, 2)

```

```

        pn_block_name => parse_node_get_sub_ptr (pn_block_spec)
        if (trim (adjustl (upper_case (parse_node_get_string &
            (pn_block_name)))) == block_name) then
            return
        end if
    end select
    pn_block => parse_node_get_next_ptr (pn_block)
end do
if (required) then
    call msg_fatal ("SLHA: block '" // char (block_name) // "' not found")
end if
end function slha_get_block_ptr

```

Scan the file for the first/next DECAY block.

*(SLHA: procedures)*+≡

```

function slha_get_first_decay_ptr (parse_tree) result (pn_decay)
    type(parse_node_t), pointer :: pn_decay
    type(parse_tree_t), intent(in) :: parse_tree
    type(parse_node_t), pointer :: pn_root
    pn_root => parse_tree_get_root_ptr (parse_tree)
    pn_decay => parse_node_get_sub_ptr (pn_root)
    do while (associated (pn_decay))
        select case (char (parse_node_get_rule_key (pn_decay)))
            case ("decay_def")
                return
        end select
        pn_decay => parse_node_get_next_ptr (pn_decay)
    end do
end function slha_get_first_decay_ptr

function slha_get_next_decay_ptr (pn_block) result (pn_decay)
    type(parse_node_t), pointer :: pn_decay
    type(parse_node_t), intent(in), target :: pn_block
    pn_decay => parse_node_get_next_ptr (pn_block)
    do while (associated (pn_decay))
        select case (char (parse_node_get_rule_key (pn_decay)))
            case ("decay_def")
                return
        end select
        pn_decay => parse_node_get_next_ptr (pn_decay)
    end do
end function slha_get_next_decay_ptr

```

## Extract and transfer data from blocks

Given the parse node of a block, find the parse node of a particular switch or data line. Return this node and the node of the data item following the integer code.

*(SLHA: procedures)*+≡

```

subroutine slha_find_index_ptr (pn_block, pn_data, pn_item, code)
    type(parse_node_t), intent(in), target :: pn_block
    ! type(parse_node_t), intent(out), pointer :: pn_data

```



```

!   type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_data
      type(parse_node_t), pointer :: pn_item
      integer, intent(in) :: code
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      call slha_next_index_ptr (pn_data, pn_item, code)
end subroutine slha_find_index_ptr

subroutine slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
      type(parse_node_t), intent(in), target :: pn_block
!   type(parse_node_t), intent(out), pointer :: pn_data
!   type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_data
      type(parse_node_t), pointer :: pn_item
      integer, intent(in) :: code1, code2
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      call slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
end subroutine slha_find_index_pair_ptr

```

Starting from the pointer to a data line, find a data line with the given integer code.

*(SLHA: procedures)*+≡

```

subroutine slha_next_index_ptr (pn_data, pn_item, code)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code
!   type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code
      do while (associated (pn_data))
         pn_line => parse_node_get_sub_ptr (pn_data, 2)
         pn_code => parse_node_get_sub_ptr (pn_line)
         select case (char (parse_node_get_rule_key (pn_code)))
            case ("integer")
               if (parse_node_get_integer (pn_code) == code) then
                  pn_item => parse_node_get_next_ptr (pn_code)
                  return
               end if
            end select
         pn_data => parse_node_get_next_ptr (pn_data)
      end do
      pn_item => null ()
end subroutine slha_next_index_ptr

```

Starting from the pointer to a data line, find a data line with the given integer code pair.

*(SLHA: procedures)*+≡

```

subroutine slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code1, code2
!   type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code1, pn_code2
      do while (associated (pn_data))

```

```

pn_line => parse_node_get_sub_ptr (pn_data, 2)
pn_code1 => parse_node_get_sub_ptr (pn_line)
select case (char (parse_node_get_rule_key (pn_code1)))
case ("integer")
  if (parse_node_get_integer (pn_code1) == code1) then
    pn_code2 => parse_node_get_next_ptr (pn_code1)
    if (associated (pn_code2)) then
      select case (char (parse_node_get_rule_key (pn_code2)))
      case ("integer")
        if (parse_node_get_integer (pn_code2) == code2) then
          pn_item => parse_node_get_next_ptr (pn_code2)
          return
        end if
      end select
    end if
  end if
end select
pn_data => parse_node_get_next_ptr (pn_data)
end do
pn_item => null ()
end subroutine slha_next_index_pair_ptr

```

## Handle info data

Return all strings with index i. The result is an allocated string array. Since we do not know the number of matching entries in advance, we build an intermediate list which is transferred to the final array and deleted before exiting.

(*SLHA: procedures*) +=

```

subroutine retrieve_strings_in_block (pn_block, code, str_array)
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code
  type(string_t), dimension(:), allocatable, intent(out) :: str_array
  type(parse_node_t), pointer :: pn_data, pn_item
  type :: str_entry_t
    type(string_t) :: str
    type(str_entry_t), pointer :: next => null ()
  end type str_entry_t
  type(str_entry_t), pointer :: first => null ()
  type(str_entry_t), pointer :: current => null ()
  integer :: n
  n = 0
  call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
  if (associated (pn_item)) then
    n = n + 1
    allocate (first)
    first%str = parse_node_get_string (pn_item)
    current => first
    do while (associated (pn_data))
      pn_data => parse_node_get_next_ptr (pn_data)
      call slha_next_index_ptr (pn_data, pn_item, code)
      if (associated (pn_item)) then
        n = n + 1

```

```

        allocate (current%next)
        current => current%next
        current%str = parse_node_get_string (pn_item)
    end if
end do
allocate (str_array (n))
n = 0
do while (associated (first))
    n = n + 1
    current => first
    str_array(n) = current%str
    first => first%next
    deallocate (current)
end do
else
    allocate (str_array (0))
end if
end subroutine retrieve_strings_in_block

```

### Transfer data from SLHA to variables

Extract real parameter with index *i*. If it does not exist, retrieve it from the variable list, using the given name.

*(SLHA: procedures)* +=

```

function get_parameter_in_block (pn_block, code, name, var_list) result (var)
    real(default) :: var
    type(parse_node_t), intent(in), target :: pn_block
    integer, intent(in) :: code
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_data, pn_item
    call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
    if (associated (pn_item)) then
        var = get_real_parameter (pn_item)
    else
        var = var_list_get_rval (var_list, name)
    end if
end function get_parameter_in_block

```

Extract a real data item with index *i*. If it does exist, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

*(SLHA: procedures)* +=

```

subroutine set_data_item (pn_block, code, name, var_list)
    type(parse_node_t), intent(in), target :: pn_block
    integer, intent(in) :: code
    type(string_t), intent(in) :: name
    type(var_list_t), intent(inout), target :: var_list
    type(parse_node_t), pointer :: pn_data, pn_item
    call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
    if (associated (pn_item)) then
        call var_list_set_real &

```

```

        (var_list, name, get_real_parameter (pn_item), &
         is_known=.true., ignore=.true.)
    end if
end subroutine set_data_item

```

Extract a real matrix element with index  $i, j$ . If it does exists, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

*(SLHA: procedures)*+≡

```

subroutine set_matrix_element (pn_block, code1, code2, name, var_list)
    type(parse_node_t), intent(in), target :: pn_block
    integer, intent(in) :: code1, code2
    type(string_t), intent(in) :: name
    type(var_list_t), intent(inout), target :: var_list
    type(parse_node_t), pointer :: pn_data, pn_item
    call slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
    if (associated (pn_item)) then
        call var_list_set_real &
            (var_list, name, get_real_parameter (pn_item), &
             is_known=.true., ignore=.true.)
    end if
end subroutine set_matrix_element

```

## Transfer data from variables to SLHA

Get a real/integer parameter with index  $i$  from the variable list and write it to the current output file. In the integer case, we account for the fact that the variable is type real. If it does not exist, do nothing.

*(SLHA: procedures)*+≡

```

subroutine write_integer_data_item (u, code, name, var_list, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in) :: var_list
    character(*), intent(in) :: comment
    integer :: item
    if (var_list_exists (var_list, name)) then
        item = nint (var_list_get_rval (var_list, name))
        call write_integer_parameter (u, code, item, comment)
    end if
end subroutine write_integer_data_item

subroutine write_real_data_item (u, code, name, var_list, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in) :: var_list
    character(*), intent(in) :: comment
    real(default) :: item
    if (var_list_exists (var_list, name)) then
        item = var_list_get_rval (var_list, name)
        call write_real_parameter (u, code, item, comment)
    end if
end subroutine write_real_data_item

```

```

        end if
    end subroutine write_real_data_item

```

Get a real data item with two integer indices from the variable list and write it to the current output file. If it does not exist, do nothing.

*<SLHA: procedures>+≡*

```

subroutine write_matrix_element (u, code1, code2, name, var_list, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code1, code2
    type(string_t), intent(in) :: name
    type(var_list_t), intent(in) :: var_list
    character(*), intent(in) :: comment
    real(default) :: item
    if (var_list_exists (var_list, name)) then
        item = var_list_get_rval (var_list, name)
        call write_real_matrix_element (u, code1, code2, item, comment)
    end if
end subroutine write_matrix_element

```

### 18.0.6 Auxiliary function

Write a block header.

*<SLHA: procedures>+≡*

```

subroutine write_block_header (u, name, comment)
    integer, intent(in) :: u
    character(*), intent(in) :: name, comment
    write (u, "(A,1x,A,3x,'#',1x,A)") "BLOCK", name, comment
end subroutine write_block_header

```

Extract a real parameter that may be defined real or integer, signed or unsigned.

*<SLHA: procedures>+≡*

```

function get_real_parameter (pn_item) result (var)
    real(default) :: var
    type(parse_node_t), intent(in), target :: pn_item
    type(parse_node_t), pointer :: pn_sign, pn_var
    integer :: sign
    select case (char (parse_node_get_rule_key (pn_item)))
    case ("signed_number")
        pn_sign => parse_node_get_sub_ptr (pn_item)
        pn_var => parse_node_get_next_ptr (pn_sign)
        select case (char (parse_node_get_key (pn_sign)))
        case ("+"); sign = +1
        case ("-"); sign = -1
        end select
    case default
        sign = +1
        pn_var => pn_item
    end select
    select case (char (parse_node_get_rule_key (pn_var)))
    case ("integer"); var = sign * parse_node_get_integer (pn_var)
    case ("real"); var = sign * parse_node_get_real (pn_var)

```

```

        end select
    end function get_real_parameter

```

Auxiliary: Extract an integer parameter that may be defined signed or unsigned.  
A real value is an error.

*<SLHA: procedures>+≡*

```

function get_integer_parameter (pn_item) result (var)
    integer :: var
    type(parse_node_t), intent(in), target :: pn_item
    type(parse_node_t), pointer :: pn_sign, pn_var
    integer :: sign
    select case (char (parse_node_get_rule_key (pn_item)))
    case ("signed_integer")
        pn_sign => parse_node_get_sub_ptr (pn_item)
        pn_var => parse_node_get_next_ptr (pn_sign)
        select case (char (parse_node_get_key (pn_sign)))
        case ("+"); sign = +1
        case ("-"); sign = -1
        end select
    case ("integer")
        sign = +1
        pn_var => pn_item
    case default
        call parse_node_write (pn_var)
        call msg_error ("SLHA: Integer parameter expected")
        var = 0
        return
    end select
    var = sign * parse_node_get_integer (pn_var)
end function get_integer_parameter

```

Write an integer parameter with a single index directly to file, using the required output format.

*<SLHA: procedures>+≡*

```

subroutine write_integer_parameter (u, code, item, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    integer, intent(in) :: item
    character(*), intent(in) :: comment
1   format (1x, I9, 3x, 3x, I9, 4x, 3x, '#', 1x, A)
    write (u, 1) code, item, comment
end subroutine write_integer_parameter

```

Write a real parameter with two indices directly to file, using the required output format.

*<SLHA: procedures>+≡*

```

subroutine write_real_parameter (u, code, item, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    real(default), intent(in) :: item
    character(*), intent(in) :: comment
1   format (1x, I9, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)

```

```

        write (u, 1) code, item, comment
    end subroutine write_real_parameter

```

Write a real parameter with a single index directly to file, using the required output format.

```

<SLHA: procedures>+=
    subroutine write_real_matrix_element (u, code1, code2, item, comment)
        integer, intent(in) :: u
        integer, intent(in) :: code1, code2
        real(default), intent(in) :: item
        character(*), intent(in) :: comment
1    format (1x, I2, 1x, I2, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
        write (u, 1) code1, code2, item, comment
    end subroutine write_real_matrix_element

```

## The concrete SLHA interpreter

SLHA codes for particular physics models

```

<SLHA: parameters>+=
    integer, parameter :: MDL_MSSM = 0
    integer, parameter :: MDL_NMSSM = 1

```

Take the parse tree and extract relevant data. Select the correct model and store all data that is present in the appropriate variable list. Finally, update the variable record.

```

<SLHA: procedures>+=
    subroutine slha_interpret_parse_tree &
        (parse_tree, os_data, model, input, spectrum, decays)
        type(parse_tree_t), intent(in) :: parse_tree
        type(os_data_t), intent(in) :: os_data
    !   type(model_t), pointer, intent(out) :: model
        type(model_t), pointer, intent(inout) :: model
        logical, intent(in) :: input, spectrum, decays
        logical :: errors
        integer :: mssm_type
        call slha_handle_MODSEL (parse_tree, os_data, model, mssm_type)
        if (associated (model)) then
            if (input) then
                call slha_handle_SMINPUTS (parse_tree, model)
                call slha_handle_MINPAR (parse_tree, model, mssm_type)
            end if
            if (spectrum) then
                call slha_handle_info_block (parse_tree, "SPINFO", errors)
                if (errors) return
                call slha_handle_MASS (parse_tree, model)
                call slha_handle_matrix_block (parse_tree, "NMIX", "mn_", 4, 4, model)
                call slha_handle_matrix_block (parse_tree, "NMNMIX", "mixn_", 5, 5, model)
                call slha_handle_matrix_block (parse_tree, "UMIX", "mu_", 2, 2, model)
                call slha_handle_matrix_block (parse_tree, "VMIX", "mv_", 2, 2, model)
                call slha_handle_matrix_block (parse_tree, "STOPMIX", "mt_", 2, 2, model)
                call slha_handle_matrix_block (parse_tree, "SBOTMIX", "mb_", 2, 2, model)
                call slha_handle_matrix_block (parse_tree, "STAUMIX", "ml_", 2, 2, model)
            end if
        end if
    end subroutine slha_interpret_parse_tree

```

```

        call slha_handle_matrix_block (parse_tree, "NMHMIX", "mixh0_", 3, 3, model)
        call slha_handle_matrix_block (parse_tree, "NMAMIX", "mixa0_", 2, 3, model)
        call slha_handle_ALPHA (parse_tree, model)
        call slha_handle_HMIX (parse_tree, model)
        call slha_handle_NMSSMRUN (parse_tree, model)
        call slha_handle_matrix_block (parse_tree, "AU", "Au_", 3, 3, model)
        call slha_handle_matrix_block (parse_tree, "AD", "Ad_", 3, 3, model)
        call slha_handle_matrix_block (parse_tree, "AE", "Ae_", 3, 3, model)
    end if
    if (decays) then
        call slha_handle_info_block (parse_tree, "DCINFO", errors)
        if (errors) return
        call slha_handle_decays (parse_tree, model)
    end if
end if
end subroutine slha_interpret_parse_tree

```

## Info blocks

Handle the informational blocks SPINFO and DCINFO. The first two items are program name and version. Items with index 3 are warnings. Items with index 4 are errors. We reproduce these as WHIZARD warnings and errors.

*(SLHA: procedures)*+≡

```

subroutine slha_handle_info_block (parse_tree, block_name, errors)
    type(parse_tree_t), intent(in) :: parse_tree
    character(*), intent(in) :: block_name
    logical, intent(out) :: errors
    type(parse_node_t), pointer :: pn_block
    type(string_t), dimension(:), allocatable :: msg
    integer :: i
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str (block_name), required=.true.)
    if (.not. associated (pn_block)) then
        call msg_error ("SLHA: Missing info block '" &
            // trim (block_name) // "'; ignored.")
        errors = .true.
        return
    end if
    select case (block_name)
    case ("SPINFO")
        call msg_message ("SLHA: SUSY spectrum program info:")
    case ("DCINFO")
        call msg_message ("SLHA: SUSY decay program info:")
    end select
    call retrieve_strings_in_block (pn_block, 1, msg)
    do i = 1, size (msg)
        call msg_message ("SLHA: " // char (msg(i)))
    end do
    call retrieve_strings_in_block (pn_block, 2, msg)
    do i = 1, size (msg)
        call msg_message ("SLHA: " // char (msg(i)))
    end do
end do

```



```

call retrieve_strings_in_block (pn_block, 3, msg)
do i = 1, size (msg)
    call msg_warning ("SLHA: " // char (msg(i)))
end do
call retrieve_strings_in_block (pn_block, 4, msg)
do i = 1, size (msg)
    call msg_error ("SLHA: " // char (msg(i)))
end do
errors = size (msg) > 0
end subroutine slha_handle_info_block

```

## MODSEL

Handle the overall model definition. Only certain models are recognized. The soft-breaking model templates that determine the set of input parameters:

*<SLHA: parameters>+≡*

```

integer, parameter :: MSSM_GENERIC = 0
integer, parameter :: MSSM_SUGRA = 1
integer, parameter :: MSSM_GMSB = 2
integer, parameter :: MSSM_AMSB = 3

```

*<SLHA: procedures>+≡*

```

subroutine slha_handle_MODSEL (parse_tree, os_data, model, mssm_type)
    type(parse_tree_t), intent(in) :: parse_tree
    type(os_data_t), intent(in) :: os_data
    type(model_t), pointer, intent(inout) :: model
    integer, intent(out) :: mssm_type
    type(parse_node_t), pointer :: pn_block, pn_data, pn_item
    type(string_t) :: model_name
    type(string_t) :: filename
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("MODSEL"), required=.true.)
    call slha_find_index_ptr (pn_block, pn_data, pn_item, 1)
    if (associated (pn_item)) then
        mssm_type = get_integer_parameter (pn_item)
    else
        mssm_type = MSSM_GENERIC
    end if
    call slha_find_index_ptr (pn_block, pn_data, pn_item, 3)
    if (associated (pn_item)) then
        select case (parse_node_get_integer (pn_item))
            case (MDL_MSSM); model_name = "MSSM"
            case (MDL_NMSSM); model_name = "NMSSM"
            case default
                call msg_fatal ("SLHA: unknown model code in MODSEL")
                return
            end select
    else
        model_name = "MSSM"
    end if
    call slha_find_index_ptr (pn_block, pn_data, pn_item, 4)
    if (associated (pn_item)) then

```

```

        call msg_fatal (" R-parity violation is currently not supported by WHIZARD.")
    end if
    call slha_find_index_ptr (pn_block, pn_data, pn_item, 5)
    if (associated (pn_item)) then
        call msg_fatal (" CP violation is currently not supported by WHIZARD.")
    end if
    select case (char (model_name))
        case ("MSSM")
            select case (char (model_get_name (model)))
                case ("MSSM","MSSM_CKM","MSSM_Grav","MSSM_Hgg")
                    model_name = model_get_name (model)
                case default
                    call msg_fatal (" User-defined model and model in SLHA input file do not match.")
            end select
        case ("NMSSM")
            select case (char (model_get_name (model)))
                case ("NMSSM","NMSSM_CKM","NMSSM_Hgg")
                    model_name = model_get_name (model)
                case default
                    call msg_fatal (" User-defined model and model in SLHA input file do not match.")
            end select
        case default
            call msg_fatal (" SLHA model selection failure.")
    end select
    filename = model_name // ".mdl"
    model => null ()
    call model_list_read_model (model_name, filename, os_data, model)
    if (associated (model)) then
        call msg_message ("SLHA: Initializing model '" &
            // char (model_name) // "'")
    else
        call msg_fatal ("SLHA: Initialization failed for model '" &
            // char (model_name) // "'")
    end if
end subroutine slha_handle_MODSEL

```

Write a MODSEL block, based on the contents of the current model.

*(SLHA: procedures)*+≡

```

subroutine slha_write_MODSEL (u, model, mssm_type)
    integer, intent(in) :: u
    type(model_t), intent(in), target :: model
    integer, intent(out) :: mssm_type
    type(var_list_t), pointer :: var_list
    integer :: model_id
    type(string_t) :: mtype_string
    var_list => model_get_var_list_ptr (model)
    if (var_list_exists (var_list, var_str ("mtype"))) then
        mssm_type = nint (var_list_get_rval (var_list, var_str ("mtype")))
    else
        call msg_error ("SLHA: parameter 'mtype' (SUSY breaking scheme) " &
            // "is unknown in current model, no SLHA output possible")
        mssm_type = -1
    return
end subroutine

```

```

end if
call write_block_header (u, "MODSEL", "SUSY model selection")
select case (mssm_type)
case (0); mtype_string = "Generic MSSM"
case (1); mtype_string = "SUGRA"
case (2); mtype_string = "GMSB"
case (3); mtype_string = "AMSB"
case default
    mtype_string = "unknown"
end select
call write_integer_parameter (u, 1, mssm_type, &
    "SUSY-breaking scheme: " // char (mtype_string))
select case (char (model_get_name (model)))
case ("MSSM"); model_id = MDL_MSSM
case ("NMSSM"); model_id = MDL_NMSSM
case default
    model_id = 0
end select
call write_integer_parameter (u, 3, model_id, &
    "SUSY model type: " // char (model_get_name (model)))
end subroutine slha_write_MODSEL

```

## SMINPUTS

Read SM parameters and update the variable list accordingly. If a parameter is not defined in the block, we use the previous value from the model variable list. For the basic parameters we have to do a small recalculation, since SLHA uses the  $G_F$ - $\alpha$ - $m_Z$  scheme, while WHIZARD derives them from  $G_F$ ,  $m_W$ , and  $m_Z$ .

(*SLHA: procedures*) +=

```

subroutine slha_handle_SMINPUTS (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    real(default) :: alpha_em_i, GF, alphas, mZ
    real(default) :: ee, vv, cw_sw, cw2, mW
    real(default) :: mb, mtop, mtau
    type(var_list_t), pointer :: var_list
    var_list => model_get_var_list_ptr (model)
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("SMINPUTS"), required=.true.)
    if (.not. (associated (pn_block))) return
    alpha_em_i = &
        get_parameter_in_block (pn_block, 1, var_str ("alpha_em_i"), var_list)
    GF = get_parameter_in_block (pn_block, 2, var_str ("GF"), var_list)
    alphas = &
        get_parameter_in_block (pn_block, 3, var_str ("alphas"), var_list)
    mZ = get_parameter_in_block (pn_block, 4, var_str ("mZ"), var_list)
    mb = get_parameter_in_block (pn_block, 5, var_str ("mb"), var_list)
    mtop = get_parameter_in_block (pn_block, 6, var_str ("mtop"), var_list)
    mtau = get_parameter_in_block (pn_block, 7, var_str ("mtau"), var_list)
    ee = sqrt (4 * pi / alpha_em_i)
    vv = 1 / sqrt (sqrt (2._default) * GF)

```

```

cw_sw = ee * vv / (2 * mZ)
if (2*cw_sw <= 1) then
  cw2 = (1 + sqrt (1 - 4 * cw_sw**2)) / 2
  mW = mZ * sqrt (cw2)
  call var_list_set_real (var_list, var_str ("GF"), GF, .true.)
  call var_list_set_real (var_list, var_str ("mZ"), mZ, .true.)
  call var_list_set_real (var_list, var_str ("mW"), mW, .true.)
  call var_list_set_real (var_list, var_str ("mtau"), mtau, .true.)
  call var_list_set_real (var_list, var_str ("mb"), mb, .true.)
  call var_list_set_real (var_list, var_str ("mtop"), mtop, .true.)
  call var_list_set_real (var_list, var_str ("alphas"), alphas, .true.)
else
  call msg_fatal ("SLHA: Unphysical SM parameter values")
  return
end if
end subroutine slha_handle_SMINPUTS

```

Write a SMINPUTS block.

*<SLHA: procedures>+≡*

```

subroutine slha_write_SMINPUTS (u, model)
  integer, intent(in) :: u
  type(model_t), intent(in), target :: model
  type(var_list_t), pointer :: var_list
  integer :: model_id
  var_list => model_get_var_list_ptr (model)
  call write_block_header (u, "SMINPUTS", "SM input parameters")
  call write_real_data_item (u, 1, var_str ("alpha_em_i"), var_list, &
    "Inverse electromagnetic coupling alpha (Z pole)")
  call write_real_data_item (u, 2, var_str ("GF"), var_list, &
    "Fermi constant")
  call write_real_data_item (u, 3, var_str ("alphas"), var_list, &
    "Strong coupling alpha_s (Z pole)")
  call write_real_data_item (u, 4, var_str ("mZ"), var_list, &
    "Z mass")
  call write_real_data_item (u, 5, var_str ("mb"), var_list, &
    "b running mass (at mb)")
  call write_real_data_item (u, 6, var_str ("mtop"), var_list, &
    "top mass")
  call write_real_data_item (u, 7, var_str ("mtau"), var_list, &
    "tau mass")
end subroutine slha_write_SMINPUTS

```

## MINPAR

The block of SUSY input parameters. They are accessible to WHIZARD, but they only get used when an external spectrum generator is invoked. The precise set of parameters depends on the type of SUSY breaking, which by itself is one of the parameters.

*<SLHA: procedures>+≡*

```

subroutine slha_handle_MINPAR (parse_tree, model, mssm_type)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(inout), target :: model

```

```

integer, intent(in) :: mssm_type
type(var_list_t), pointer :: var_list
type(parse_node_t), pointer :: pn_block
var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, &
    var_str ("mtype"), real(mssm_type, default), is_known=.true.)
pn_block => slha_get_block_ptr &
    (parse_tree, var_str ("MINPAR"), required=.true.)
select case (mssm_type)
case (MSSM_SUGRA)
    call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
    call set_data_item (pn_block, 2, var_str ("m_half"), var_list)
    call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
    call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
    call set_data_item (pn_block, 5, var_str ("A0"), var_list)
case (MSSM_GMSB)
    call set_data_item (pn_block, 1, var_str ("Lambda"), var_list)
    call set_data_item (pn_block, 2, var_str ("M_mes"), var_list)
    call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
    call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
    call set_data_item (pn_block, 5, var_str ("N_5"), var_list)
    call set_data_item (pn_block, 6, var_str ("c_grav"), var_list)
case (MSSM_AMSB)
    call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
    call set_data_item (pn_block, 2, var_str ("m_grav"), var_list)
    call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
    call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
case default
    call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
end select
end subroutine slha_handle_MINPAR

```

Write a MINPAR block as appropriate for the current model type.

*(SLHA: procedures)*+≡

```

subroutine slha_write_MINPAR (u, model, mssm_type)
integer, intent(in) :: u
type(model_t), intent(in), target :: model
integer, intent(in) :: mssm_type
type(var_list_t), pointer :: var_list
integer :: model_id
var_list => model_get_var_list_ptr (model)
call write_block_header (u, "MINPAR", "Basic SUSY input parameters")
select case (mssm_type)
case (MSSM_SUGRA)
    call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
        "Common scalar mass")
    call write_real_data_item (u, 2, var_str ("m_half"), var_list, &
        "Common gaugino mass")
    call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
        "tan(beta)")
    call write_integer_data_item (u, 4, &
        var_str ("sgn_mu"), var_list, &
        "Sign of mu")
    call write_real_data_item (u, 5, var_str ("A0"), var_list, &

```

```

        "Common trilinear coupling")
case (MSSM_GMSB)
    call write_real_data_item (u, 1, var_str ("Lambda"), var_list, &
        "Soft-breaking scale")
    call write_real_data_item (u, 2, var_str ("M_mes"), var_list, &
        "Messenger scale")
    call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
        "tan(beta)")
    call write_integer_data_item (u, 4, &
        var_str ("sgn_mu"), var_list, &
        "Sign of mu")
    call write_integer_data_item (u, 5, var_str ("N_5"), var_list, &
        "Messenger index")
    call write_real_data_item (u, 6, var_str ("c_grav"), var_list, &
        "Gravitino mass factor")
case (MSSM_AMSB)
    call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
        "Common scalar mass")
    call write_real_data_item (u, 2, var_str ("m_grav"), var_list, &
        "Gravitino mass")
    call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
        "tan(beta)")
    call write_integer_data_item (u, 4, &
        var_str ("sgn_mu"), var_list, &
        "Sign of mu")
case default
    call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
        "tan(beta)")
end select
end subroutine slha_write_MINPAR

```

## Mass spectrum

Set masses. Since the particles are identified by PDG code, read the line and try to set the appropriate particle mass in the current model. At the end, update parameters, just in case the  $W$  or  $Z$  mass was included.

(*SLHA: procedures*) +=

```

subroutine slha_handle_MASS (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block, pn_data, pn_line, pn_code
    type(parse_node_t), pointer :: pn_mass
    integer :: pdg
    real(default) :: mass
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("MASS"), required=.true.)
    if (.not. (associated (pn_block))) return
    pn_data => parse_node_get_sub_ptr (pn_block, 4)
    do while (associated (pn_data))
        pn_line => parse_node_get_sub_ptr (pn_data, 2)
        pn_code => parse_node_get_sub_ptr (pn_line)
        if (associated (pn_code)) then

```

```

    pdg = get_integer_parameter (pn_code)
    pn_mass => parse_node_get_next_ptr (pn_code)
    if (associated (pn_mass)) then
        mass = get_real_parameter (pn_mass)
        call model_set_particle_mass (model, pdg, mass)
    else
        call msg_error ("SLHA: Block MASS: Missing mass value")
    end if
else
    call msg_error ("SLHA: Block MASS: Missing PDG code")
end if
pn_data => parse_node_get_next_ptr (pn_data)
end do
end subroutine slha_handle_MASS

```

## Widths

Set widths. For each DECAY block, extract the header, read the PDG code and width, and try to set the appropriate particle width in the current model.

*(SLHA: procedures)*+≡

```

subroutine slha_handle_decays (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_decay, pn_decay_spec, pn_code, pn_width
    integer :: pdg
    real(default) :: width
    pn_decay => slha_get_first_decay_ptr (parse_tree)
    do while (associated (pn_decay))
        pn_decay_spec => parse_node_get_sub_ptr (pn_decay, 2)
        pn_code => parse_node_get_sub_ptr (pn_decay_spec)
        pdg = get_integer_parameter (pn_code)
        pn_width => parse_node_get_next_ptr (pn_code)
        width = get_real_parameter (pn_width)
        call model_set_particle_width (model, pdg, width)
        pn_decay => slha_get_next_decay_ptr (pn_decay)
    end do
end subroutine slha_handle_decays

```

## Mixing matrices

Read mixing matrices. We can treat all matrices by a single procedure if we just know the block name, variable prefix, and matrix dimension. The matrix dimension must be less than 10. For the pseudoscalar Higgses in NMSSM-type models we need off-diagonal matrices, so we generalize the definition.

*(SLHA: procedures)*+≡

```

subroutine slha_handle_matrix_block &
    (parse_tree, block_name, var_prefix, dim1, dim2, model)
    type(parse_tree_t), intent(in) :: parse_tree
    character(*), intent(in) :: block_name, var_prefix
    integer, intent(in) :: dim1, dim2
    type(model_t), intent(inout), target :: model

```

```

type(parse_node_t), pointer :: pn_block
type(var_list_t), pointer :: var_list
integer :: i, j
character(len=len(var_prefix)+2) :: var_name
var_list => model_get_var_list_ptr (model)
pn_block => slha_get_block_ptr &
    (parse_tree, var_str (block_name), required=.false.)
if (.not. (associated (pn_block))) return
do i = 1, dim1
    do j = 1, dim2
        write (var_name, "(A,I1,I1)") var_prefix, i, j
        call set_matrix_element (pn_block, i, j, var_str (var_name), var_list)
    end do
end do
end subroutine slha_handle_matrix_block

```

## Higgs data

Read the block ALPHA which holds just the Higgs mixing angle.

*(SLHA: procedures)+≡*

```

subroutine slha_handle_ALPHA (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block, pn_line, pn_data, pn_item
    type(var_list_t), pointer :: var_list
    real(default) :: al_h
    var_list => model_get_var_list_ptr (model)
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("ALPHA"), required=.false.)
    if (.not. (associated (pn_block))) return
    pn_data => parse_node_get_sub_ptr (pn_block, 4)
    pn_line => parse_node_get_sub_ptr (pn_data, 2)
    pn_item => parse_node_get_sub_ptr (pn_line)
    if (associated (pn_item)) then
        al_h = get_real_parameter (pn_item)
        call var_list_set_real (var_list, var_str ("al_h"), al_h, &
            is_known=.true., ignore=.true.)
    end if
end subroutine slha_handle_ALPHA

```

Read the block HMX for the Higgs mixing parameters

*(SLHA: procedures)+≡*

```

subroutine slha_handle_HMX (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    type(var_list_t), pointer :: var_list
    var_list => model_get_var_list_ptr (model)
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("HMX"), required=.false.)
    if (.not. (associated (pn_block))) return
    call set_data_item (pn_block, 1, var_str ("mu_h"), var_list)

```



```

        call set_data_item (pn_block, 2, var_str ("tanb_h"), var_list)
    end subroutine slha_handle_HMIX

```

Read the block NMSSMRUN for the specific NMSSM parameters

*(SLHA: procedures)*+≡

```

subroutine slha_handle_NMSSMRUN (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    type(var_list_t), pointer :: var_list
    var_list => model_get_var_list_ptr (model)
    pn_block => slha_get_block_ptr &
        (parse_tree, var_str ("NMSSMRUN"), required=.false.)
    if (.not. (associated (pn_block))) return
    call set_data_item (pn_block, 1, var_str ("ls"), var_list)
    call set_data_item (pn_block, 2, var_str ("ks"), var_list)
    call set_data_item (pn_block, 3, var_str ("a_ls"), var_list)
    call set_data_item (pn_block, 4, var_str ("a_ks"), var_list)
    call set_data_item (pn_block, 5, var_str ("nmu"), var_list)
end subroutine slha_handle_NMSSMRUN

```

## 18.0.7 Parser

Read a SLHA file from stream, including preprocessing, and make up a parse tree.

*(SLHA: procedures)*+≡

```

subroutine slha_parse_stream (stream, parse_tree)
    type(stream_t), intent(inout), target :: stream
    type(parse_tree_t), intent(out) :: parse_tree
    type(ifile_t) :: ifile
    type(lexer_t) :: lexer
    type(stream_t), target :: stream_tmp
    call slha_preprocess (stream, ifile)
    call stream_init (stream_tmp, ifile)
    call lexer_init_slha (lexer)
    call lexer_assign_stream (lexer, stream_tmp)
    call parse_tree_init (parse_tree, syntax_slha, lexer)
    call lexer_final (lexer)
    call stream_final (stream_tmp)
    call ifile_final (ifile)
end subroutine slha_parse_stream

```

Read a SLHA file chosen by name. Check first the current directory, then the directory where SUSY input files should be located.

*(SLHA: procedures)*+≡

```

subroutine slha_parse_file (file, os_data, parse_tree)
    type(string_t), intent(in) :: file
    type(os_data_t), intent(in) :: os_data
    type(parse_tree_t), intent(out) :: parse_tree
    logical :: exist
    type(string_t) :: filename

```

```

type(stream_t), target :: stream
call msg_message ("Reading SLHA input file '" // char (file) // "'")
filename = file
inquire (file=char(filename), exist=exist)
if (.not. exist) then
  filename = os_data%whizard_susypath // "/" // file
  inquire (file=char(filename), exist=exist)
  if (.not. exist) then
    call msg_fatal ("SLHA input file '" // char (file) // "' not found")
    return
  end if
end if
call stream_init (stream, char (filename))
call slha_parse_stream (stream, parse_tree)
call stream_final (stream)
end subroutine slha_parse_file

```

## 18.0.8 API

Read the SLHA file, parse it, and interpret the parse tree. The model parameters retrieved from the file will be inserted into the appropriate model, which is loaded and modified in the background. The pointer to this model is returned as the last argument.

```

<SLHA: public>+≡
  public :: slha_read_file

<SLHA: procedures>+≡
  subroutine slha_read_file (file, os_data, model, input, spectrum, decays)
    type(string_t), intent(in) :: file
    type(os_data_t), intent(in) :: os_data
    ! type(model_t), pointer, intent(out) :: model
    type(model_t), pointer, intent(inout) :: model
    logical, intent(in) :: input, spectrum, decays
    type(parse_tree_t) :: parse_tree
    call slha_parse_file (file, os_data, parse_tree)
    if (associated (parse_tree_get_root_ptr (parse_tree))) then
      call slha_interpret_parse_tree &
        (parse_tree, os_data, model, input, spectrum, decays)
      call parse_tree_final (parse_tree)
      call model_parameters_update (model)
    end if
  end subroutine slha_read_file

```

Write the SLHA contents, as far as possible, to external file.

```

<SLHA: public>+≡
  public :: slha_write_file

<SLHA: procedures>+≡
  subroutine slha_write_file (file, model, input, spectrum, decays)
    type(string_t), intent(in) :: file
    type(model_t), target, intent(in) :: model
    logical, intent(in) :: input, spectrum, decays

```

```

integer :: mssm_type
integer :: u
u = free_unit ()
call msg_message ("Writing SLHA output file '" // char (file) // "'")
open (unit=u, file=char(file), action="write", status="replace")
write (u, "(A)")  "# SUSY Les Houches Accord"
write (u, "(A)")  "# Output generated by " // trim (VERSION_STRING)
call slha_write_MODSEL (u, model, mssm_type)
if (input) then
    call slha_write_SMINPUTS (u, model)
    call slha_write_MINPAR (u, model, mssm_type)
end if
if (spectrum) then
    call msg_bug ("SLHA: spectrum output not supported yet")
end if
if (decays) then
    call msg_bug ("SLHA: decays output not supported yet")
end if
close (u)
end subroutine slha_write_file

```

## 18.0.9 Test

*(SLHA: public)*+≡

```
public :: slha_test
```

*(SLHA: procedures)*+≡

```

subroutine slha_test ()
    type(os_data_t), pointer :: os_data => null ()
    type(parse_tree_t), pointer :: parse_tree => null ()
    integer :: unit
    character(*), parameter :: file_test = "slha_test.out"
    character(*), parameter :: file_slha = "slha_test.dat"
    type(model_t), pointer :: model => null ()
    allocate (os_data)
    allocate (parse_tree)
    call os_data_init (os_data)
    call model_list_read_model (var_str("MSSM"), var_str("MSSM.mdl"), os_data, model)
    call slha_parse_file (var_str ("spslap_decays.slha"), os_data, parse_tree)
    call msg_message ("Writing parse tree to '" // file_test // "'")
    unit = free_unit ()
    open (unit=unit, file=file_test, action="write", status="replace")
    call parse_tree_write (parse_tree, unit)
    call slha_interpret_parse_tree (parse_tree, os_data, model, &
        input=.true., spectrum=.true., decays=.true.)
    call parse_tree_final (parse_tree)
    call var_list_write (model_get_var_list_ptr (model), only_type=V_REAL)
    call msg_message ("Writing SLHA output to '" // file_slha // "'")
    call slha_write_file (var_str (file_slha), model, input=.true., &
        spectrum=.false., decays=.false.)
    call parse_tree_final (parse_tree)
    deallocate (parse_tree)
    deallocate (os_data)

```

```
end subroutine slha_test
```

## Chapter 19

# Integration and simulation

This layer of modules is just below the top-level API. We lay out specific data types for integration and simulation and implement the corresponding algorithms as methods acting on them. This helps to keep the command-level implementation concise and simple.

### 19.1 Iterations

This module defines a container for the list of iterations and calls, to be submitted to integration.

```
<iterations.f90>≡  
  <File header>  
  
  module iterations  
  
    <Use strings>  
    <Use file utils>  
    use limits, only: ITERATIONS_DEFAULT_LIST_SIZE !NODEP!  
    use diagnostics !NODEP!  
    use unit_tests  
    use processes  
  
    <Standard module head>  
  
    <Iterations: public>  
  
    <Iterations: types>  
  
    <Iterations: interfaces>  
  
    contains  
  
    <Iterations: procedures>  
  
    <Iterations: tests>  
  
  end module iterations
```

### 19.1.1 The iterations list

Each integration pass has a number of iterations and a number of calls per iteration. The last pass produces the end result; the previous passes are used for adaptation.

The flags `adapt_grid` and `adapt_weight` are used only if `custom_adaptation` is set. Otherwise, default settings are used that depend on the integration pass.

```
<Iterations: types>≡
  type :: iterations_spec_t
  private
    integer :: n_it = 0
    integer :: n_calls = 0
    logical :: custom_adaptation = .false.
    logical :: adapt_grids = .false.
    logical :: adapt_weights = .false.
  end type iterations_spec_t
```

We build up a list of iterations.

```
<Iterations: public>≡
  public :: iterations_list_t
```

```
<Iterations: types>+≡
  type :: iterations_list_t
  private
    integer :: n_pass = 0
    type(iterations_spec_t), dimension(:), allocatable :: pass
  contains
    <Iterations: iterations list: TBP>
  end type iterations_list_t
```

```
<Iterations: iterations list: TBP>≡
  procedure :: init => iterations_list_init
```

```
<Iterations: procedures>≡
  subroutine iterations_list_init (it_list, n_it, n_calls, adapt, adapt_code)
    class(iterations_list_t), intent(inout) :: it_list
    integer, dimension(:), intent(in) :: n_it, n_calls
    logical, dimension(:), intent(in), optional :: adapt
    type(string_t), dimension(:), intent(in), optional :: adapt_code
    it_list%n_pass = size (n_it)
    if (allocated (it_list%pass)) deallocate (it_list%pass)
    allocate (it_list%pass (it_list%n_pass))
    it_list%pass%n_it = n_it
    it_list%pass%n_calls = n_calls
    if (present (adapt)) then
      it_list%pass%custom_adaptation = adapt
      if (any (verify (adapt_code, "wg") /= 0)) then
        call msg_error ("iteration specification: " &
          // "adaptation code letters must be 'w' or 'g'")
      end if
      it_list%pass%adapt_grids = scan (adapt_code, "g") /= 0
      it_list%pass%adapt_weights = scan (adapt_code, "w") /= 0
    end if
  end subroutine iterations_list_init
```

Fill all zero entries by corresponding entries from a default iterations list:

```

<CCC Iterations: public>≡
    public :: iterations_list_complete

<CCC Iterations: procedures>≡
    subroutine iterations_list_complete (it_list, it_list_default)
        type(iterations_list_t), intent(inout) :: it_list
        type(iterations_list_t), intent(in) :: it_list_default
        if (it_list%n_pass >= 1) then
            if (it_list%pass(1)%n_it == 0) &
                it_list%pass(1)%n_it = it_list_default%pass(1)%n_it
            if (it_list%pass(1)%n_calls == 0) &
                it_list%pass(1)%n_calls = it_list_default%pass(1)%n_calls
        end if
        if (it_list%n_pass >= 2) then
            where (it_list%pass%n_it == 0) &
                it_list%pass%n_it = it_list_default%pass(2)%n_it
            where (it_list%pass%n_calls == 0) &
                it_list%pass%n_calls = it_list_default%pass(2)%n_calls
        end if
    end subroutine iterations_list_complete

<Iterations: iterations list: TBP>+≡
    procedure :: clear => iterations_list_clear

<Iterations: procedures>+≡
    subroutine iterations_list_clear (it_list)
        class(iterations_list_t), intent(inout) :: it_list
        it_list%n_pass = 0
        deallocate (it_list%pass)
    end subroutine iterations_list_clear

```

Write the list of iterations as a message.

```

<Iterations: iterations list: TBP>+≡
    procedure :: write => iterations_list_write

<Iterations: procedures>+≡
    subroutine iterations_list_write (it_list, unit)
        class(iterations_list_t), intent(in) :: it_list
        integer, intent(in), optional :: unit
        integer :: u
        type(string_t) :: buffer
        character(30) :: ibuf
        integer :: i
        u = output_unit (unit)
        buffer = "iterations = "
        if (it_list%n_pass > 0) then
            do i = 1, it_list%n_pass
                if (i > 1) buffer = buffer // ", "
                write (ibuf, "(I0,':',I0)") &
                    it_list%pass(i)%n_it, it_list%pass(i)%n_calls
                buffer = buffer // trim (ibuf)
                if (it_list%pass(i)%custom_adaptation) then

```

```

        buffer = buffer // ':'
        if (it_list%pass(i)%adapt_grids) buffer = buffer // "g"
        if (it_list%pass(i)%adapt_weights) buffer = buffer // "w"
        buffer = buffer // ','
    end if
end do
else
    buffer = buffer // "[undefined]"
end if
!    call msg_message (char (buffer), unit)
    write (u, "(A)") char (buffer)
end subroutine iterations_list_write

```

### 19.1.2 Tools

Transform the iterations list into arrays that indicate pass index and number of calls for each iteration.

*<CCC Iterations: public>+≡*

```

    public :: iterations_list_get_pass_array
    public :: iterations_list_get_n_calls_array

```

*<CCC Iterations: procedures>+≡*

```

function iterations_list_get_pass_array (it_list) result (pass)
    integer, dimension(:), allocatable :: pass
    type(iterations_list_t), intent(in) :: it_list
    integer :: it, i
    allocate (pass (sum (it_list%pass%n_it)))
    it = 0
    do i = 1, it_list%n_pass
        pass(it+1 : it+it_list%pass(i)%n_it) = i
        it = it + it_list%pass(i)%n_it
    end do
end function iterations_list_get_pass_array

function iterations_list_get_n_calls_array (it_list) result (n_calls)
    integer, dimension(:), allocatable :: n_calls
    type(iterations_list_t), intent(in) :: it_list
    integer :: it, i
    allocate (n_calls (sum (it_list%pass%n_it)))
    it = 0
    do i = 1, it_list%n_pass
        n_calls(it+1 : it+it_list%pass(i)%n_it) = it_list%pass(i)%n_calls
        it = it + it_list%pass(i)%n_it
    end do
end function iterations_list_get_n_calls_array

```

Return the total number of passes.

*<Iterations: iterations list: TBP>+≡*

```

    procedure :: get_n_pass => iterations_list_get_n_pass

```

*<Iterations: procedures>+≡*

```

function iterations_list_get_n_pass (it_list) result (n_pass)
    class(iterations_list_t), intent(in) :: it_list

```



```

integer :: n_pass
n_pass = it_list%n_pass
end function iterations_list_get_n_pass

```

Return the number of calls for a specific pass.

*<Iterations: iterations list: TBP>+≡*

```

procedure :: get_n_calls => iterations_list_get_n_calls

```

*<Iterations: procedures>+≡*

```

function iterations_list_get_n_calls (it_list, pass) result (n_calls)
class(iterations_list_t), intent(in) :: it_list
integer :: n_calls
integer, intent(in) :: pass
if (pass <= it_list%n_pass) then
    n_calls = it_list%pass(pass)%n_calls
else
    n_calls = 0
end if
end function iterations_list_get_n_calls

```

Get the adaptation mode (automatic/custom) and, for custom adaptation, the flags for a specific pass.

*<CCC Iterations: public>+≡*

```

public :: iterations_list_has_custom_adaptation
public :: iterations_list_adapt_grids
public :: iterations_list_adapt_weights

```

*<Iterations: iterations list: TBP>+≡*

```

procedure :: adapt_grids => iterations_list_adapt_grids
procedure :: adapt_weights => iterations_list_adapt_weights

```

*<Iterations: procedures>+≡*

```

! function iterations_list_has_custom_adaptation (it_list, pass) result (flag)
!     logical :: flag
!     type(iterations_list_t), intent(in) :: it_list
!     integer, intent(in) :: pass
!     if (pass <= it_list%n_pass) then
!         flag = it_list%pass(pass)%custom_adaptation
!     else
!         flag = .false.
!     end if
! end function iterations_list_has_custom_adaptation

function iterations_list_adapt_grids (it_list, pass) result (flag)
logical :: flag
class(iterations_list_t), intent(in) :: it_list
integer, intent(in) :: pass
if (pass <= it_list%n_pass) then
    flag = it_list%pass(pass)%adapt_grids
else
    flag = .false.
end if
end function iterations_list_adapt_grids

```

```

function iterations_list_adapt_weights (it_list, pass) result (flag)
  logical :: flag
  class(iterations_list_t), intent(in) :: it_list
  integer, intent(in) :: pass
  if (pass <= it_list%n_pass) then
    flag = it_list%pass(pass)%adapt_weights
  else
    flag = .false.
  end if
end function iterations_list_adapt_weights

```

Return the total number of iterations / the iterations for a specific pass.

*<Iterations: iterations list: TBP>+≡*

```

  procedure :: get_n_it => iterations_list_get_n_it

```

*<CCC Iterations: interfaces>≡*

```

  interface iterations_list_get_n_it
    module procedure iterations_list_get_n_it_tot
    module procedure iterations_list_get_n_it_pass
  end interface

```

*<Iterations: procedures>+≡*

```

!   function iterations_list_get_n_it_tot (it_list) result (n_it)
!     integer :: n_it
!     type(iterations_list_t), intent(in) :: it_list
!     n_it = sum (it_list%pass%n_it)
!   end function iterations_list_get_n_it_tot

```

```

function iterations_list_get_n_it (it_list, pass) result (n_it)
  class(iterations_list_t), intent(in) :: it_list
  integer :: n_it
  integer, intent(in) :: pass
  if (pass <= it_list%n_pass) then
    n_it = it_list%pass(pass)%n_it
  else
    n_it = 0
  end if
end function iterations_list_get_n_it

```

This routine corrects for a number of calls that is too low.

*<CCC Iterations: public>+≡*

```

  public :: iterations_list_adjust_n_calls

```

*<CCC Iterations: procedures>+≡*

```

  subroutine iterations_list_adjust_n_calls (it_list, process, grid_parameters)
    type(iterations_list_t), intent(inout), target :: it_list
    type(process_t), intent(in) :: process
    type(grid_parameters_t), intent(in) :: grid_parameters
    type(iterations_spec_t), pointer :: it_spec
    integer :: n_calls, pass
    logical :: changed
    changed = .false.
    do pass = 1, it_list%n_pass

```

```

        it_spec => it_list%pass(pass)
        n_calls = max (it_spec%n_calls, &
            process_get_n_channels (process) &
            * grid_parameters%min_calls_per_channel)
        if (n_calls /= it_spec%n_calls) then
            it_spec%n_calls = n_calls
            changed = .true.
        end if
    end do
    if (changed) then
        write (msg_buffer, "(A,I0)") "Process '" &
            // char (process_get_id (process)) // "' : " &
            // "resetting n_calls to ", n_calls
        call msg_warning ()
    end if
end subroutine iterations_list_adjust_n_calls

```

Allocate the default iterations lists and fill with some sensible values. This is implemented as a pointer since it is not intended to be modified by the user; the local RT dataset will thus contain another pointer to the same list, not a copy.

```

<Limits: public parameters>+=
    integer, parameter, public :: ITERATIONS_DEFAULT_LIST_SIZE = 7

<CCC Iterations: public>+=
    public :: iterations_lists_init_default

<CCC Iterations: procedures>+=
    subroutine iterations_lists_init_default (it_list)
        type(iterations_list_t), dimension(:), pointer :: it_list
        allocate (it_list (ITERATIONS_DEFAULT_LIST_SIZE))
        call iterations_list_init (it_list(1), (/ 1 /), (/ 100 /))
        call iterations_list_init (it_list(2), (/ 3, 3 /), (/ 1000, 10000 /))
        call iterations_list_init (it_list(3), (/ 5, 3 /), (/ 5000, 10000 /))
        call iterations_list_init (it_list(4), (/ 10, 5 /), (/ 10000, 20000 /))
        call iterations_list_init (it_list(5), (/ 10, 5 /), (/ 20000, 50000 /))
        call iterations_list_init (it_list(6), (/ 15, 5 /), (/ 50000, 100000 /))
        call iterations_list_init (it_list(7), (/ 20, 5 /), (/ 50000, 200000 /))
    end subroutine iterations_lists_init_default

```

### 19.1.3 Test

This is the master for calling self-test procedures.

```

<Iterations: public>+=
    public :: iterations_test

<Iterations: tests>=
    subroutine iterations_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
        <Iterations: execute tests>
    end subroutine iterations_test

```

## Empty list

```
<Iterations: execute tests>≡
  call test (iterations_1, "iterations_1", &
    "empty iterations list", &
    u, results)

<Iterations: tests>+≡
  subroutine iterations_1 (u)
    integer, intent(in) :: u
    type(iterations_list_t) :: it_list

    write (u, "(A)")  "* Test output: iterations_1"
    write (u, "(A)")  "* Purpose: display empty iterations list"
    write (u, "(A)")

    call it_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: iterations_1"

  end subroutine iterations_1
```

## Fill list

```
<Iterations: execute tests>+≡
  call test (iterations_2, "iterations_2", &
    "create iterations list", &
    u, results)

<Iterations: tests>+≡
  subroutine iterations_2 (u)
    integer, intent(in) :: u
    type(iterations_list_t) :: it_list

    write (u, "(A)")  "* Test output: iterations_2"
    write (u, "(A)")  "* Purpose: fill and display iterations list"
    write (u, "(A)")

    write (u, "(A)")  "* Minimal setup (2 passes)"
    write (u, "(A)")

    call it_list%init ([2, 4], [5000, 20000])

    call it_list%write (u)
    call it_list%clear ()

    write (u, "(A)")
    write (u, "(A)")  "* Setup with flags (3 passes)"
    write (u, "(A)")

    call it_list%init ([2, 4, 5], [5000, 20000, 400], &
      [.false., .true., .true.], &
      [var_str (""), var_str ("g"), var_str ("wg")])
```

```

call it_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract data"
write (u, "(A)")

write (u, "(A,I0)")  "n_pass = ", it_list%get_n_pass ()
write (u, "(A)")
write (u, "(A,I0)")  "n_calls(2) = ", it_list%get_n_calls (2)
write (u, "(A)")
write (u, "(A,I0)")  "n_it(3) = ", it_list%get_n_it (3)

write (u, "(A)")
write (u, "(A)")  "* Test output end: iterations_2"

end subroutine iterations_2

```

## 19.2 Beam polarization

Beam polarization is encapsulated in a designated type and only converted into a density matrix when the beams are actually initialized — the flavor information necessary for initialization is not available earlier.

```

⟨beam_polarizations.f90⟩≡
  ⟨File header⟩

  module beam_polarizations

    ⟨Use kinds⟩
    ⟨Use strings⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use flavors
    use polarizations

    ⟨Standard module head⟩

    ⟨Beam polarizations: public⟩

    ⟨Beam polarizations: parameters⟩

    ⟨Beam polarizations: types⟩

    contains

    ⟨Beam polarizations: procedures⟩

  end module beam_polarizations

```

## 19.2.1 Parameters and type definition

```
<Beam polarizations: public>≡
  public :: BP_NONE, BP_CIRC, BP_TRANS, BP_LONG, BP_AXIS, BP_DIAG, BP_DENSITY
  public :: BP_TRIVIAL
  public :: beam_polarization_t

<Beam polarizations: parameters>≡
  integer, parameter :: &
    BP_NONE = 0, BP_CIRC = 1, BP_TRANS = 2, BP_LONG = 3, BP_AXIS = 4, &
    BP_DIAG = 5, BP_DENSITY = 6, BP_TRIVIAL = 7

<Beam polarizations: types>≡
  type :: beam_polarization_t
  private
  integer :: type = BP_NONE
  real(default) :: fraction
  real(default) :: theta
  real(default) :: phi
  real(default) :: d
  complex(default) :: nd
  integer, dimension(:), allocatable :: hels
  real(default), dimension(:), allocatable :: fractions
end type beam_polarization_t
```

## 19.2.2 Constructors

The type is filled by dedicated constructors:

```
<Beam polarizations: public>+≡
  public :: beam_polarization_init_none
  public :: beam_polarization_init_trivial
  public :: beam_polarization_init_circ
  public :: beam_polarization_init_trans
  public :: beam_polarization_init_long
  public :: beam_polarization_init_axis
  public :: beam_polarization_init_diag
  public :: beam_polarization_init_density
  public :: beam_polarization_final

<Beam polarizations: procedures>≡
  subroutine beam_polarization_init_none (bp)
    type(beam_polarization_t), intent(inout) :: bp
    bp%type = BP_NONE
  end subroutine beam_polarization_init_none

  subroutine beam_polarization_init_trivial (bp)
    type(beam_polarization_t), intent(inout) :: bp
    bp%type = BP_TRIVIAL
  end subroutine beam_polarization_init_trivial

  subroutine beam_polarization_init_circ (bp, fraction)
    type(beam_polarization_t), intent(inout) :: bp
    real(default), intent(in) :: fraction
    bp%type = BP_CIRC
```

```

        bp%fraction = fraction
    end subroutine beam_polarization_init_circ

subroutine beam_polarization_init_trans (bp, fraction, phi)
    type(beam_polarization_t), intent(inout) :: bp
    real(default), intent(in) :: fraction, phi
    bp%type = BP_TRANS
    bp%fraction = fraction
    bp%phi = phi
end subroutine beam_polarization_init_trans

subroutine beam_polarization_init_long (bp, fraction)
    type(beam_polarization_t), intent(inout) :: bp
    real(default), intent(in) :: fraction
    bp%type = BP_LONG
    bp%fraction = fraction
end subroutine beam_polarization_init_long

subroutine beam_polarization_init_axis (bp, fraction, theta, phi)
    type(beam_polarization_t), intent(inout) :: bp
    real(default), intent(in) :: fraction, theta, phi
    bp%type = BP_AXIS
    bp%fraction = fraction
    bp%theta = theta
    bp%phi = phi
end subroutine beam_polarization_init_axis

subroutine beam_polarization_init_diag (bp, hels, fracs)
    type(beam_polarization_t), intent(inout) :: bp
    integer, dimension(:), intent(in) :: hels
    real(default), dimension(:), intent(in) :: fracs
    bp%type = BP_DIAG
    allocate (bp%hels(size (hels)))
    allocate (bp%fractions(size (fracs)))
    bp%hels = hels
    bp%fractions = fracs
end subroutine beam_polarization_init_diag

subroutine beam_polarization_init_density (bp, d, nd)
    type(beam_polarization_t), intent(inout) :: bp
    real(default), intent(in) :: d
    complex(default), intent(in) :: nd
    bp%type = BP_DENSITY
    bp%d = d
    bp%nd = nd
end subroutine beam_polarization_init_density

subroutine beam_polarization_final (bp)
    type(beam_polarization_t), intent(inout) :: bp
    if (allocated (bp%hels)) deallocate (bp%hels)
    if (allocated (bp%fractions)) deallocate (bp%fractions)
end subroutine beam_polarization_final

```

### 19.2.3 Tools

Together with the necessary flavor information, `beam_polarization_t` can be promoted to `polarization_t`

```

(Beam polarizations: public) +=
    public :: beam_polarization2polarization

(Beam polarizations: procedures) +=
    function beam_polarization2polarization (bp, flv, decay) result (pol)
        type(beam_polarization_t), intent(in) :: bp
        type(flavor_t), intent(in) :: flv
        logical, optional, intent(in) :: decay
        type(polarization_t) :: pol
        logical :: fail
        real(default), dimension(:), allocatable :: frac_vector
        integer :: i, j, mult
        type(string_t) :: msg
        if (flavor_get_multiplicity (flv) == 1) then
            select case (bp%type)
                case (BP_NONE, BP_TRIVIAL)
                case default
                    if (flavor_is_left_handed (flv)) then
                        msg = "left-handed"
                    elseif (flavor_is_right_handed (flv)) then
                        msg = "right-handed"
                    else
                        msg = "scalar"
                    end if
                    call msg_error (char (msg) // " particle '" &
                        // char (flavor_get_name (flv)) &
                        // "' cannot be polarized - ignoring polarization")
                    call emergency_unpolarized
                    return
            end select
        end if
        select case (bp%type)
            case (BP_NONE)
                call polarization_init_unpolarized (pol, flv)
            case (BP_TRIVIAL)
                call polarization_init_trivial (pol, flv)
            case (BP_CIRC)
                if ((bp%fraction <= 1) .and. (bp%fraction >= -1)) then
                    call polarization_init_circular (pol, flv, bp%fraction)
                else
                    call msg_error ( &
                        "circular polarization: 'fraction' must be within [-1; 1] - " &
                        // "ignoring polarization")
                    call emergency_unpolarized
                end if
            case (BP_TRANS)
                if ((bp%fraction <= 1) .and. (bp%fraction >= -1)) then
                    call polarization_init_transversal (pol, flv, bp%phi, bp%fraction)
                else
                    call msg_error ( &
                        "transverse polarization: 'fraction' must be within [-1; 1] - " &

```



```

        // "ignoring polarization")
        call emergency_unpolarized
    end if
case (BP_LONG)
    if ((bp%fraction > 1) .or. (bp%fraction < 0)) then
        call msg_error ( &
            "longitudinal polarization: 'fraction' must be within [0; 1]" &
            // " - ignoring polarization");
        call emergency_unpolarized
    elseif (mod (flavor_get_multiplicity (flv), 2) == 0) then
        call msg_error ( &
            "longitudinal polarization is only available for massive " &
            // " bosons - ignoring polarization")
        call emergency_unpolarized
    else
        call polarization_init_longitudinal (pol, flv, bp%fraction)
    end if
case (BP_AXIS)
    if ((bp%fraction <= 1) .and. (bp%fraction >= -1)) then
        call polarization_init_angles (pol, flv, bp%fraction, bp%theta, &
            bp%phi)
    else
        call msg_error ( &
            "axial polarization: 'fraction' must be within [-1; 1] - " &
            // "ignoring polarization")
        call emergency_unpolarized
    end if
case (BP_DENSITY)
    if ((bp%d <= 1) .and. (bp%d >= 0) .and. (abs (bp%nd) <= 0.5)) then
        call polarization_init_axis (pol, flv, &
            (/real (bp%nd, default), (-1.) * aimag (bp%nd), 2. * bp%d - 1.))
    else
        call msg_error ( &
            "density matrix polarization: 'a' must be within [0; 1], |b| " &
            // "within [0; 0.5] - ignoring polarization")
        call emergency_unpolarized
    end if
case (BP_DIAG)
    fail = .false.
    mult = flavor_get_multiplicity (flv)
    allocate (frac_vector (mult))
    frac_vector = 0
    if (minval (bp%fractions) < 0) then
        call msg_error ( &
            "diagonal polarization: negative fractions are not allowed " &
            // "- ignoring polarization")
        fail = .true.
    else
        select case (mult)
        case (1)
            call msg_bug ( &
                "beam_polarization2polarization: invalid multiplicity")
        case (2)
            if ((size (bp%hels) <= 2) .and. all (abs (bp%hels) == 1)) then

```

```

        frac_vector = 0
        do i = 1, size(bp%hels)
            frac_vector((bp%hels(i) + 1) / 2 + 1) = bp%fractions(i)
        end do
    else
        call msg_error ( &
            "diagonal polarization: the only admissible helicities " &
            // "for particle '" // char (flavor_get_name (flv)) &
            // "' are" // " -1 and 1 - ignoring polarization")
        fail = .true.
    end if
case default
    if (maxval (abs (bp%hels)) <= mult / 2) then
        if (mod (mult, 2) == 0) then
            if (minval (abs (bp%hels)) == 0) then
                call msg_error ( &
                    "diagonal polarization: helicity 0 not allowed " &
                    // "for particle '" // char (flavor_get_name (flv)) &
                    // "' - ignoring polarization")
                fail = .true.
            else
                do i = 1, size (bp%hels)
                    if (bp%hels(i) < 0) then
                        j = bp%hels(i) + mult / 2 + 1
                    else
                        j = bp%hels(i) + mult / 2
                    end if
                    frac_vector(j) = bp%fractions(i)
                end do
            end if
        else
            do i = 1, size (bp%hels)
                j = bp%hels(i) + mult / 2 + 1
                frac_vector(j) = bp%fractions(i)
            end do
        end if
    else
        call msg_error ( &
            "diagonal polarization: helicity exceeds admissible " &
            // "range for particle '" // char (flavor_get_name (flv)) &
            // "' - ignoring polarization")
        fail = .true.
    end if
end select
end if
if (fail) then
    call emergency_unpolarized
else
    if (sum (frac_vector) /= 1) &
        call msg_warning ( &
            "diagonal polarization: fractions will be normalized to 1")
    call polarization_init_diagonal (pol, flv, frac_vector)
end if
deallocate (frac_vector)

```

```

end select

contains

subroutine emergency_unpolarized
  logical :: is_decay
  if (present (decay)) then
    is_decay = decay
  else
    is_decay = .false.
  end if
  if (is_decay) then
    call polarization_init_trivial (pol, flv)
  else
    call polarization_init_unpolarized (pol, flv)
  end if
end subroutine emergency_unpolarized

end function beam_polarization2polarization

```

Writing.

```

<Beam polarizations: public>+≡
  public :: beam_polarization_write

<Beam polarizations: procedures>+≡
  subroutine beam_polarization_write (bp, unit, indent)
    type (beam_polarization_t), intent (in) :: bp
    integer, intent (in), optional :: unit, indent
    integer :: u, i
    type (string_t), dimension (:), allocatable :: msgs
    type (string_t) :: header, is
    u = output_unit (unit)
    if (u < 0) return
    select case (bp%type)
      case (BP_NONE, BP_TRIVIAL)
        call printer ("none")
      case (BP_CIRC)
        call printer ("circular (fraction):")
        call printer ("  fraction: " // real2char (bp%fraction))
      case (BP_TRANS)
        call printer ("transverse (fraction, phi):")
        call printer ("  fraction: " // real2char (bp%fraction))
        call printer ("  phi      : " // real2char (bp%phi))
      case (BP_AXIS)
        call printer ("axis (fraction, theta, phi):")
        call printer ("  fraction: " // real2char (bp%fraction))
        call printer ("  theta   : " // real2char (bp%theta))
        call printer ("  phi     : " // real2char (bp%phi))
      case (BP_LONG)
        call printer ("longitudinal (fraction):")
        call printer ("  fraction: " // real2char (bp%fraction))
      case (BP_DENSITY)
        call printer ("density_matrix (a, b):")
        call printer ("  a: " // real2char (bp%d))
    end select
  end subroutine beam_polarization_write

```

```

        call printer ("    b: " // char (cmplx2string (bp%nd)))
    case (BP_DIAG)
        allocate (msgs(size (bp%fractions)))
        header = "diagonal_density ("
        do i = 1, size (msgs)
            is = int2string (i)
            if (i > 1) header = header // ", "
            header = header // "h" // is // ":f" // is
            msgs (i) = "h" // is // ": " // int2string (bp%hels(i)) &
                // " , f" // is // ": " // real2string (bp%fractions(i))
        end do
        call printer (char (header) // ")")
        do i = 1, size (msgs)
            call printer ("    " // char (msgs(i)))
        end do
        deallocate (msgs)
    case default
        call msg_bug ("beam_polarization_write: illegal polarization type")
    end select
    flush (u)

contains

    subroutine printer (s)
        character(*), intent(in) :: s
        if (present (indent)) write (u, '(A)', advance="no") &
            repeat (" ", indent)
        write (u, '(1x,A)') s
    end subroutine printer

end subroutine beam_polarization_write

```

## 19.3 Beam structure

This module stores the beam structure definition as it is declared in the SIN-DARIN script. The structure definition is not analyzed, just recorded for later use.

```

⟨beam_structures.f90⟩≡
  ⟨File header⟩

  module beam_structures

    ⟨Use strings⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use unit_tests
    use variables

    ⟨Standard module head⟩

    ⟨Beam structures: public⟩

```

```

    <Beam structures: types>

    <Beam structures: interfaces>

    contains

    <Beam structures: procedures>

    <Beam structures: tests>

    end module beam_structures

```

### 19.3.1 Beam structure elements

An entry in a beam-structure record consists of a string that denotes a type of structure function.

```

<Beam structures: types>≡
    type :: beam_structure_entry_t
        logical :: is_valid = .false.
        type(string_t) :: name
    contains
        <Beam structures: beam structure entry: TBP>
    end type beam_structure_entry_t

```

Output.

```

<Beam structures: beam structure entry: TBP>≡
    procedure :: to_string => beam_structure_entry_to_string

<Beam structures: procedures>≡
    function beam_structure_entry_to_string (object) result (string)
        class (beam_structure_entry_t), intent(in) :: object
        type(string_t) :: string
        if (object%is_valid) then
            string = object%name
        else
            string = "none"
        end if
    end function beam_structure_entry_to_string

```

A record in the beam-structure sequence denotes either a structure-function entry, a pair of such entries, or a pair spectrum.

```

<Beam structures: types>+≡
    type :: beam_structure_record_t
        type (beam_structure_entry_t), dimension(:), allocatable :: entry
    end type beam_structure_record_t

```

### 19.3.2 Beam structure type

The beam-structure object is a sequence of records.

```

<Beam structures: public>≡

```

```

    public :: beam_structure_t
<Beam structures: types>+≡
    type :: beam_structure_t
        type (beam_structure_record_t), dimension(:), allocatable :: record
        type (var_list_t) :: var_list
    contains
        <Beam structures: beam structure: TBP>
    end type beam_structure_t

```

Output. The actual information fits in a single line, therefore we can provide a `to_string` method.

```

<Beam structures: beam structure: TBP>≡
    procedure :: write => beam_structure_write
    procedure :: to_string => beam_structure_to_string
<Beam structures: procedures>+≡
    subroutine beam_structure_write (object, unit)
        class (beam_structure_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        if (allocated (object%record)) then
            if (size (object%record) > 0) then
                write (u, "(1x,A)") "Beam structure:"
                write (u, "(3x,A)") char (object%to_string ())
            else
                write (u, "(1x,A)") "Beam structure: [trivial]"
            end if
        else
            write (u, "(1x,A)") "Beam structure: [undefined]"
        end if
    end subroutine beam_structure_write

    function beam_structure_to_string (object) result (string)
        class (beam_structure_t), intent(in) :: object
        type (string_t) :: string
        integer :: i, j
        if (allocated (object%record)) then
            string = ""
            if (size (object%record) > 0) then
                do i = 1, size (object%record)
                    if (i > 1) string = string // " => "
                    do j = 1, size (object%record(i)%entry)
                        if (j > 1) string = string // ", "
                        string = string // object%record(i)%entry(j)%to_string ()
                    end do
                end do
            else
                string = "[trivial]"
            end if
        else
            string = "[undefined]"
        end if
    end function beam_structure_to_string

```

Initializer: dimension the beam structure record. Each array element denotes the number of entries for a record within the beam-structure sequence. The number of entries is either one or two, while the number of records is unlimited.

```

<Beam structures: beam structure: TBP>+=
  procedure :: init => beam_structure_init

<Beam structures: procedures>+=
  subroutine beam_structure_init (beam_structure, dim_array, var_list)
    class(beam_structure_t), intent(out) :: beam_structure
    integer, dimension(:), intent(in) :: dim_array
    type(var_list_t), intent(in) :: var_list
    integer :: i
    allocate (beam_structure%record (size (dim_array)))
    do i = 1, size (dim_array)
      allocate (beam_structure%record(i)%entry (dim_array(i)))
    end do
    call var_list_init_snapshot (beam_structure%var_list, var_list)
  end subroutine beam_structure_init

```

Set an entry, specified by record number and entry number.

```

<Beam structures: beam structure: TBP>+=
  procedure :: set_entry => beam_structure_set_entry

<Beam structures: procedures>+=
  subroutine beam_structure_set_entry (beam_structure, i, j, name)
    class(beam_structure_t), intent(inout) :: beam_structure
    integer, intent(in) :: i, j
    type(string_t), intent(in) :: name
    associate (entry => beam_structure%record(i)%entry(j))
      entry%name = name
      entry%is_valid = .true.
    end associate
  end subroutine beam_structure_set_entry

```

Expand the beam-structure object. (i) For a pair spectrum, keep the entry. (ii) For a single-particle structure function written as a single entry, replace this by a record with two entries. (ii) For a record with two nontrivial entries, separate this into two records with one trivial entry each.

To achieve this, we need a function that tells us whether an entry is a spectrum or a structure function. It returns 0 for a trivial entry, 1 for a single-particle structure function, and 2 for a two-particle spectrum.

```

<Beam structures: interfaces>=
  abstract interface
    function strfun_mode_fun (name) result (n)
      import
      type(string_t), intent(in) :: name
      integer :: n
    end function strfun_mode_fun
  end interface

```

Algorithm: (1) Mark entries as invalid where necessary. (2) Count the number of entries that we will need. (3) Expand and copy entries to a new record array. (4) Replace the old array by the new one.

*(Beam structures: beam structure: TBP)+≡*

procedure :: expand => beam\_structure\_expand

*(Beam structures: procedures)+≡*

```

subroutine beam_structure_expand (beam_structure, strfun_mode)
  class(beam_structure_t), intent(inout) :: beam_structure
  procedure(strfun_mode_fun) :: strfun_mode
  type(beam_structure_record_t), dimension(:), allocatable :: new
  integer :: n_record, i, j
  if (.not. allocated (beam_structure%record)) return
  do i = 1, size (beam_structure%record)
    associate (entry => beam_structure%record(i)%entry)
      do j = 1, size (entry)
        select case (strfun_mode (entry(j)%name))
          case (0); entry(j)%is_valid = .false.
        end select
      end do
    end associate
  end do
  n_record = 0
  do i = 1, size (beam_structure%record)
    associate (entry => beam_structure%record(i)%entry)
      select case (size (entry))
        case (1)
          if (entry(1)%is_valid) then
            select case (strfun_mode (entry(1)%name))
              case (1); n_record = n_record + 2
              case (2); n_record = n_record + 1
            end select
          end if
        case (2)
          do j = 1, 2
            if (entry(j)%is_valid) then
              select case (strfun_mode (entry(j)%name))
                case (1); n_record = n_record + 1
                case (2)
                  call beam_structure%write ()
                  call msg_fatal ("Pair spectrum used as &
                                &single-particle structure function")
              end select
            end if
          end do
        end select
      end associate
    end do
  allocate (new (n_record))
  n_record = 0
  do i = 1, size (beam_structure%record)
    associate (entry => beam_structure%record(i)%entry)
      select case (size (entry))
        case (1)

```



```

        if (entry(1)%is_valid) then
        select case (strfun_mode (entry(1)%name))
        case (1)
            n_record = n_record + 1
            allocate (new(n_record)%entry (2))
            new(n_record)%entry(1) = entry(1)
            n_record = n_record + 1
            allocate (new(n_record)%entry (2))
            new(n_record)%entry(2) = entry(1)
        case (2)
            n_record = n_record + 1
            allocate (new(n_record)%entry (1))
            new(n_record)%entry(1) = entry(1)
        end select
        end if
    case (2)
        do j = 1, 2
            if (entry(j)%is_valid) then
                n_record = n_record + 1
                allocate (new(n_record)%entry (2))
                new(n_record)%entry(j) = entry(j)
            end if
        end do
    end select
end associate
end do
call move_alloc (from = new, to = beam_structure%record)
end subroutine beam_structure_expand

```

### 19.3.3 Get contents

Return the number of records.

```

(Beam structures: beam structure: TBP)+≡
    procedure :: get_n_record => beam_structure_get_n_record

(Beam structures: procedures)+≡
    function beam_structure_get_n_record (beam_structure) result (n)
        class(beam_structure_t), intent(in) :: beam_structure
        integer :: n
        if (allocated (beam_structure%record)) then
            n = size (beam_structure%record)
        else
            n = 0
        end if
    end function beam_structure_get_n_record

```

Return an array consisting of the beam indices affected by the valid entries within a record. After expansion, there should be exactly one valid entry per record.

```

(Beam structures: beam structure: TBP)+≡
    procedure :: get_i_entry => beam_structure_get_i_entry

```

```

<Beam structures: procedures>+≡
function beam_structure_get_i_entry (beam_structure, i) result (i_entry)
  class(beam_structure_t), intent(in) :: beam_structure
  integer, intent(in) :: i
  integer, dimension(:), allocatable :: i_entry
  integer :: j
  associate (record => beam_structure%record(i))
    select case (size (record%entry))
    case (1)
      if (record%entry(1)%is_valid) then
        allocate (i_entry (2), source = [1, 2])
      else
        allocate (i_entry (0))
      end if
    case (2)
      if (all (record%entry%is_valid)) then
        allocate (i_entry (2), source = [1, 2])
      else if (record%entry(1)%is_valid) then
        allocate (i_entry (1), source = [1])
      else if (record%entry(2)%is_valid) then
        allocate (i_entry (1), source = [2])
      else
        allocate (i_entry (0))
      end if
    end select
  end associate
end function beam_structure_get_i_entry

```

Return the name of the first valid entry within a record. After expansion, there should be exactly one valid entry per record.

```

<Beam structures: beam structure: TBP>+≡
procedure :: get_name => beam_structure_get_name

<Beam structures: procedures>+≡
function beam_structure_get_name (beam_structure, i) result (name)
  class(beam_structure_t), intent(in) :: beam_structure
  integer, intent(in) :: i
  type(string_t) :: name
  associate (record => beam_structure%record(i))
    if (record%entry(1)%is_valid) then
      name = record%entry(1)%name
    else if (size (record%entry) == 2) then
      name = record%entry(2)%name
    end if
  end associate
end function beam_structure_get_name

```

### 19.3.4 Unit Tests

```

<Beam structures: public>+≡
public :: beam_structures_test

```

```

<Beam structures: tests>≡
  subroutine beam_structures_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <Beam structures: execute tests>
  end subroutine beam_structures_test

```

## Empty structure

```

<Beam structures: execute tests>≡
  call test (beam_structures_1, "beam_structures_1", &
    "empty beam structure record", &
    u, results)

<Beam structures: tests>+≡
  subroutine beam_structures_1 (u)
    integer, intent(in) :: u
    type(beam_structure_t) :: beam_structure

    write (u, "(A)")  "* Test output: beam_structures_1"
    write (u, "(A)")  "* Purpose: display empty beam structure record"
    write (u, "(A)")

    call beam_structure%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_structures_1"

  end subroutine beam_structures_1

```

## Nontrivial configurations

```

<Beam structures: execute tests>+≡
  call test (beam_structures_2, "beam_structures_2", &
    "beam structure records", &
    u, results)

<Beam structures: tests>+≡
  subroutine beam_structures_2 (u)
    integer, intent(in) :: u
    type(beam_structure_t) :: beam_structure
    integer, dimension(0) :: empty_array
    type(var_list_t) :: var_list

    write (u, "(A)")  "* Test output: beam_structures_2"
    write (u, "(A)")  "* Purpose: setup beam structure records"
    write (u, "(A)")

    call var_list_append_int (var_list, var_str ("foo"), 42)

    call beam_structure%init (empty_array, var_list)
    call beam_structure%write (u)

```

```

write (u, "(A)")
write (u, "(1x,A)" "Variables:")
call var_list_write (beam_structure%var_list, u)

write (u, "(A)")

call beam_structure%init ([1], var_list)
call beam_structure%set_entry (1, 1, var_str ("a"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%init ([2], var_list)
call beam_structure%set_entry (1, 1, var_str ("a"))
call beam_structure%set_entry (1, 2, var_str ("b"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%init ([2, 1], var_list)
call beam_structure%set_entry (1, 1, var_str ("a"))
call beam_structure%set_entry (1, 2, var_str ("b"))
call beam_structure%set_entry (2, 1, var_str ("c"))
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)" " * Test output end: beam_structures_2"

end subroutine beam_structures_2

```

## Expansion

Provide a function that tells, for the dummy structure function names used here, whether they are considered a two-particle spectrum or a single-particle structure function:

```

<Beam structures: tests>+≡
function test_strfun_mode (name) result (n)
  type(string_t), intent(in) :: name
  integer :: n
  select case (char (name))
    case ("a"); n = 2
    case ("b"); n = 1
    case default; n = 0
  end select
end function test_strfun_mode

<Beam structures: execute tests>+≡
call test (beam_structures_3, "beam_structures_3", &
  "beam structure expansion", &
  u, results)

```

```

(Beam structures: tests) +=
subroutine beam_structures_3 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure
  type(var_list_t) :: var_list

  write (u, "(A)")  "* Test output: beam_structures_3"
  write (u, "(A)")  "* Purpose: expand beam structure records"
  write (u, "(A)")

  write (u, "(A)")  "* Pair spectrum (keep as-is)"
  write (u, "(A)")

  call beam_structure%init ([1], var_list)
  call beam_structure%set_entry (1, 1, var_str ("a"))
  call beam_structure%write (u)

  write (u, "(A)")

  call beam_structure%expand (test_strfun_mode)
  call beam_structure%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Structure function pair (expand)"
  write (u, "(A)")

  call beam_structure%init ([2], var_list)
  call beam_structure%set_entry (1, 1, var_str ("b"))
  call beam_structure%set_entry (1, 2, var_str ("b"))
  call beam_structure%write (u)

  write (u, "(A)")

  call beam_structure%expand (test_strfun_mode)
  call beam_structure%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Structure function (separate and expand)"
  write (u, "(A)")

  call beam_structure%init ([1], var_list)
  call beam_structure%set_entry (1, 1, var_str ("b"))
  call beam_structure%write (u)

  write (u, "(A)")

  call beam_structure%expand (test_strfun_mode)
  call beam_structure%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Combination"
  write (u, "(A)")

  call beam_structure%init ([1, 1], var_list)

```

```

call beam_structure%set_entry (1, 1, var_str ("a"))
call beam_structure%set_entry (2, 1, var_str ("b"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%expand (test_strfun_mode)
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: beam_structures_3"

end subroutine beam_structures_3

```

## Public methods

Check the methods that can be called to get the beam-structure contents.

```

<Beam structures: execute tests>+≡
  call test (beam_structures_4, "beam_structures_4", &
    "beam structure contents", &
    u, results)

<Beam structures: tests>+≡
subroutine beam_structures_4 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure
  type(var_list_t) :: var_list
  integer :: i

  write (u, "(A)")  "* Test output: beam_structures_4"
  write (u, "(A)")  "* Purpose: check the API"
  write (u, "(A)")

  write (u, "(A)")  "* Structure-function combination"
  write (u, "(A)")

  call beam_structure%init ([1, 2, 2], var_list)
  call beam_structure%set_entry (1, 1, var_str ("a"))
  call beam_structure%set_entry (2, 1, var_str ("b"))
  call beam_structure%set_entry (3, 2, var_str ("c"))
  call beam_structure%write (u)

  write (u, "(A)")
  write (u, "(1x,A,I0)")  "n_record = ", beam_structure%get_n_record ()

  do i = 1, 3
    write (u, "(A)")
    write (u, "(1x,A,I0,A,A)")  "name(", i, ") = ", &
      char (beam_structure%get_name (i))
    write (u, "(1x,A,I0,A,2(1x,I0))")  "i_entry(", i, ") = ", &
      beam_structure%get_i_entry (i)
  end do

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: beam_structures_4"

end subroutine beam_structures_4

```

## 19.4 User-controlled File I/O

The SINDARIN language includes commands that write output to file (input may be added later). We identify files by their name, and manage the unit internally. We need procedures for opening, closing, and printing files.

```

⟨user_files.f90⟩≡
  ⟨File header⟩

  module user_files

    ⟨Use strings⟩
    ⟨Use file utils⟩
    use diagnostics !NODEP!
    use ifiles
    use analysis

    ⟨Standard module head⟩

    ⟨User files: public⟩

    ⟨User files: types⟩

    ⟨User files: interfaces⟩

    contains

    ⟨User files: procedures⟩

  end module user_files

```

### 19.4.1 The file type

This is a type that describes an open user file and its properties. The entry is part of a doubly-linked list.

```

⟨User files: types⟩≡
  type :: file_t
    private
    type(string_t) :: name
    integer :: unit = -1
    logical :: reading = .false.
    logical :: writing = .false.
    type(file_t), pointer :: prev => null ()
    type(file_t), pointer :: next => null ()
  end type file_t

```

The initializer opens the file.

```
<User files: procedures>+=
subroutine file_init (file, name, action, status, position)
  type(file_t), intent(out) :: file
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action, status, position
  file%unit = free_unit ()
  file%name = name
  open (unit = file%unit, file = char (file%name), &
        action = action, status = status, position = position)
  select case (action)
  case ("read")
    file%reading = .true.
  case ("write")
    file%writing = .true.
  case ("readwrite")
    file%reading = .true.
    file%writing = .true.
  end select
end subroutine file_init
```

The finalizer closes it.

```
<User files: procedures>+=
subroutine file_final (file)
  type(file_t), intent(inout) :: file
  close (unit = file%unit)
  file%unit = -1
end subroutine file_final
```

Check if a file is open with correct status.

```
<User files: procedures>+=
function file_is_open (file, action) result (flag)
  logical :: flag
  type(file_t), intent(in) :: file
  character(*), intent(in) :: action
  select case (action)
  case ("read")
    flag = file%reading
  case ("write")
    flag = file%writing
  case ("readwrite")
    flag = file%reading .and. file%writing
  case default
    call msg_bug ("Checking file '" // char (file%name) &
                  // "': illegal action specifier")
  end select
end function file_is_open
```

Write to the file. Error if in wrong mode. If there is no string, just write an empty record. If there is a string, respect the `advancing` option.

```
<User files: procedures>+=
subroutine file_write_string (file, string, advancing)
```



```

type(file_t), intent(in) :: file
type(string_t), intent(in), optional :: string
logical, intent(in), optional :: advancing
if (file%writing) then
  if (present (string)) then
    if (present (advancing)) then
      if (advancing) then
        write (file%unit, "(A)") char (string)
      else
        write (file%unit, "(A)", advance="no") char (string)
      end if
    else
      write (file%unit, "(A)") char (string)
    end if
  else
    write (file%unit, *)
  end if
else
  call msg_error ("Writing to file: File '" // char (file%name) &
    // "' is not open for writing.")
end if
end subroutine file_write_string

```

Write a whole ifile, line by line.

```

<User files: procedures>+≡
subroutine file_write_ifile (file, ifile)
  type(file_t), intent(in) :: file
  type(ifile_t), intent(in) :: ifile
  type(line_p) :: line
  call line_init (line, ifile)
  do while (line_is_associated (line))
    call file_write_string (file, line_get_string_advance (line))
  end do
end subroutine file_write_ifile

```

Write an analysis object (or all objects) to an open file.

```

<User files: procedures>+≡
subroutine file_write_analysis (file, tag)
  type(file_t), intent(in) :: file
  type(string_t), intent(in), optional :: tag
  if (file%writing) then
    if (present (tag)) then
      call analysis_write (tag, unit = file%unit)
    else
      call analysis_write (unit = file%unit)
    end if
  else
    call msg_error ("Writing analysis to file: File '" // char (file%name) &
      // "' is not open for writing.")
  end if
end subroutine file_write_analysis

```

### 19.4.2 The file list

We maintain a list of all open files and their attributes. The list must be doubly-linked because we may delete entries.

```
<User files: public>≡
    public :: file_list_t

<User files: types>+≡
    type :: file_list_t
        type(file_t), pointer :: first => null ()
        type(file_t), pointer :: last => null ()
    end type file_list_t
```

There is no initialization routine, but a finalizer which deletes all:

```
<User files: public>+≡
    public :: file_list_final

<User files: procedures>+≡
    subroutine file_list_final (file_list)
        type(file_list_t), intent(inout) :: file_list
        type(file_t), pointer :: current
        do while (associated (file_list%first))
            current => file_list%first
            file_list%first => current%next
            call file_final (current)
            deallocate (current)
        end do
        file_list%last => null ()
    end subroutine file_list_final
```

Find an entry in the list. Return null pointer on failure.

```
<User files: procedures>+≡
    function file_list_get_file_ptr (file_list, name) result (current)
        type(file_t), pointer :: current
        type(file_list_t), intent(in) :: file_list
        type(string_t), intent(in) :: name
        current => file_list%first
        do while (associated (current))
            if (current%name == name) return
            current => current%next
        end do
    end function file_list_get_file_ptr
```

Check if a file is open, public version:

```
<User files: public>+≡
    public :: file_list_is_open

<User files: procedures>+≡
    function file_list_is_open (file_list, name, action) result (flag)
        logical :: flag
        type(file_list_t), intent(in) :: file_list
        type(string_t), intent(in) :: name
        character(len=*), intent(in) :: action
        type(file_t), pointer :: current
```

```

current => file_list_get_file_ptr (file_list, name)
if (associated (current)) then
    flag = file_is_open (current, action)
else
    flag = .false.
end if
end function file_list_is_open

```

Append a new file entry, i.e., open this file. Error if it is already open.

```

<User files: public>+≡
    public :: file_list_open

<User files: procedures>+≡
    subroutine file_list_open (file_list, name, action, status, position)
        type(file_list_t), intent(inout) :: file_list
        type(string_t), intent(in) :: name
        character(len=*), intent(in) :: action, status, position
        type(file_t), pointer :: current
        if (.not. associated (file_list_get_file_ptr (file_list, name))) then
            allocate (current)
            call file_init (current, name, action, status, position)
            if (associated (file_list%last)) then
                file_list%last%next => current
                current%prev => file_list%last
            else
                file_list%first => current
            end if
            file_list%last => current
        else
            call msg_error ("Opening file: File '" // char (name) &
                // "' is already open.")
        end if
    end subroutine file_list_open

```

Delete a file entry, i.e., close this file. Error if it is not open.

```

<User files: public>+≡
    public :: file_list_close

<User files: procedures>+≡
    subroutine file_list_close (file_list, name)
        type(file_list_t), intent(inout) :: file_list
        type(string_t), intent(in) :: name
        type(file_t), pointer :: current
        current => file_list_get_file_ptr (file_list, name)
        if (associated (current)) then
            if (associated (current%prev)) then
                current%prev%next => current%next
            else
                file_list%first => current%next
            end if
            if (associated (current%next)) then
                current%next%prev => current%prev
            else
                file_list%last => current%prev
            end if
        end if
    end subroutine file_list_close

```

```

        end if
        call file_final (current)
        deallocate (current)
    else
        call msg_error ("Closing file: File '" // char (name) &
            // "' is not open.")
    end if
end subroutine file_list_close

```

Write a string to file. Error if it is not open.

```

<User files: public>+≡
    public :: file_list_write

<User files: interfaces>≡
    interface file_list_write
        module procedure file_list_write_string
        module procedure file_list_write_ifile
    end interface

<User files: procedures>+≡
    subroutine file_list_write_string (file_list, name, string, advancing)
        type(file_list_t), intent(in) :: file_list
        type(string_t), intent(in) :: name
        type(string_t), intent(in), optional :: string
        logical, intent(in), optional :: advancing
        type(file_t), pointer :: current
        current => file_list_get_file_ptr (file_list, name)
        if (associated (current)) then
            call file_write_string (current, string, advancing)
        else
            call msg_error ("Writing to file: File '" // char (name) &
                // "'is not open.")
        end if
    end subroutine file_list_write_string

    subroutine file_list_write_ifile (file_list, name, ifile)
        type(file_list_t), intent(in) :: file_list
        type(string_t), intent(in) :: name
        type(ifile_t), intent(in) :: ifile
        type(line_p) :: line
        type(file_t), pointer :: current
        current => file_list_get_file_ptr (file_list, name)
        if (associated (current)) then
            call file_write_ifile (current, ifile)
        else
            call msg_error ("Writing to file: File '" // char (name) &
                // "'is not open.")
        end if
    end subroutine file_list_write_ifile

```

Write an analysis object or all objects to data file. Error if it is not open. If the file name is empty, write to standard output.

```

<User files: public>+≡
    public :: file_list_write_analysis

```

```

<User files: procedures>+=
subroutine file_list_write_analysis (file_list, name, tag)
  type(file_list_t), intent(in) :: file_list
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: tag
  type(file_t), pointer :: current
  if (name == "") then
    if (present (tag)) then
      call analysis_write (tag)
    else
      call analysis_write
    end if
  else
    current => file_list_get_file_ptr (file_list, name)
    if (associated (current)) then
      call file_write_analysis (current, tag)
    else
      call msg_error ("Writing analysis to file: File ' " // char (name) &
        // "' is not open.")
    end if
  end if
end subroutine file_list_write_analysis

```

## 19.5 Runtime data

```

<rt_data.f90>=
<File header>

module rt_data

  <Use kinds>
  <Use strings>
  <Use file utils>
  use system_dependencies !NODEP!
  use diagnostics !NODEP!
  ! use tao_random_numbers !NODEP!
  use unit_tests

  use pdf_builtin !NODEP!
  use sf_lhapdf !NODEP!
  use os_interface
  use ifiles
  use lexers
  use parser
  use models
  use flavors
  use variables
  use expressions
  use beams
  use process_libraries
  use prclib_stacks
  use prc_core

```

```

! use beam_polarizations
use beam_structures
! use user_files
! use nlo_setup
use process_stacks
use iterations

```

*⟨Standard module head⟩*

*⟨RT data: public⟩*

*⟨RT data: types⟩*

contains

*⟨RT data: procedures⟩*

*⟨RT data: tests⟩*

end module rt\_data

### 19.5.1 Container for parse nodes

The runtime data set contains a bunch of parse nodes (chunks of code that have not been compiled into evaluation trees but saved for later use). We collect them here.

This implementation has the useful effect that an assignment between two objects of this type will establish a pointer-target relationship for all components.

```

⟨RT data: types⟩≡
  type :: rt_parse_nodes_t
    type(parse_node_t), pointer :: cuts_lexpr => null ()
    type(parse_node_t), pointer :: scale_expr => null ()
    type(parse_node_t), pointer :: fac_scale_expr => null ()
    type(parse_node_t), pointer :: ren_scale_expr => null ()
    type(parse_node_t), pointer :: weight_expr => null ()
    type(parse_node_t), pointer :: selection_lexpr => null ()
    type(parse_node_t), pointer :: reweight_expr => null ()
    type(parse_node_t), pointer :: analysis_lexpr => null ()
!   type(parse_node_t), pointer :: histogram_writer => null ()
!   type(parse_node_t), pointer :: plot_writer => null ()
!   type(parse_node_t), pointer :: alpha_qed_expr => null ()
  contains
    ⟨RT data: rt parse nodes: TBP⟩
  end type rt_parse_nodes_t

```

Output for the parse nodes.

```

⟨RT data: rt parse nodes: TBP⟩≡
  procedure :: write => rt_parse_nodes_write

⟨RT data: procedures⟩≡
  subroutine rt_parse_nodes_write (object, unit)
    class(rt_parse_nodes_t), intent(in) :: object

```

```

integer, intent(in), optional :: unit
integer :: u
u = output_unit (unit)
call wrt ("Cuts", object%cuts_lexpr)
call write_separator (u)
call wrt ("Scale", object%scale_expr)
call write_separator (u)
call wrt ("Factorization scale", object%fac_scale_expr)
call write_separator (u)
call wrt ("Renormalization scale", object%ren_scale_expr)
call write_separator (u)
call wrt ("Weight", object%weight_expr)
call write_separator_double (u)
call wrt ("Event selection", object%selection_lexpr)
call write_separator (u)
call wrt ("Event reweighting factor", object%reweight_expr)
call write_separator (u)
call wrt ("Event analysis", object%analysis_lexpr)
!   call write_separator (u)
!   call wrt ("Histogram writer", object%histogram_writer)
!   call write_separator (u)
!   call wrt ("Plot writer", object%plot_writer)
!   call write_separator (u)
!   call wrt ("Alpha(QED)", object%alpha_qed_lexpr)
contains
subroutine wrt (title, pn)
character(*), intent(in) :: title
type(parse_node_t), intent(in), pointer :: pn
if (associated (pn)) then
write (u, "(1x,A,':')") title
call write_separator (u)
call parse_node_write_rec (pn, u)
else
write (u, "(1x,A,':',1x,A)") title, "[undefined]"
end if
end subroutine wrt
end subroutine rt_parse_nodes_write

```

### 19.5.2 The data type

This is a big data container which contains everything that is used and modified during the command flow. A local copy of this can be used to temporarily override defaults. The data set is transparent.

```

<RT data: public>≡
public :: rt_data_t

<RT data: types>+≡
type :: rt_data_t
type(lexer_t), pointer :: lexer => null ()
type(var_list_t) :: var_list
type(iterations_list_t) :: it_list
!   type(iterations_list_t), dimension(:), pointer :: it_list_default
type(os_data_t) :: os_data

```

```

type(model_t), pointer :: model => null ()
type(prclib_stack_t) :: prclib_stack
type(process_library_t), pointer :: prclib => null ()
type(beam_data_t) :: beam_data
!   type(beam_polarization_t), dimension(:), pointer :: &
!   beam_polarization => null ()
type(beam_structure_t) :: beam_structure
type(pdf_builtin_status_t) :: pdf_builtin_status
type(lhapdf_status_t) :: lhpdf_status
type(rt_parse_nodes_t) :: pn
type(process_stack_t) :: process_stack
type(string_t), dimension(:), allocatable :: sample_fmt
!   type(file_list_t), pointer :: out_files => null ()
logical :: quit = .false.
integer :: quit_code = 0
!   integer :: environment = -1
!   integer :: analysis_data_unit = -1
!   type(nlo_setup_list_t) :: nlo_setup_list
contains
  <RT data: rt data: TBP>
end type rt_data_t

```

### 19.5.3 Output

```

<RT data: rt data: TBP>≡
  procedure :: write => rt_data_write

<RT data: procedures>+≡
  subroutine rt_data_write (object, unit, vars)
    class(rt_data_t), intent(in) :: object
    integer, intent(in), optional :: unit
    type(string_t), dimension(:), intent(in), optional :: vars
    integer :: u, i
    u = output_unit (unit)
    call write_separator_double (u)
    write (u, "(1x,A)") "Runtime data:"
    if (present (vars)) then
      if (size (vars) /= 0) then
        call write_separator_double (u)
        write (u, "(1x,A)") "Selected variables:"
        call write_separator (u)
        call object%write_vars (u, vars)
      end if
    else
      call write_separator_double (u)
      call var_list_write (object%var_list, u, follow_link=.true.)
    end if
    if (object%it_list%get_n_pass () > 0) then
      call write_separator_double (u)
      write (u, "(1x)", advance="no")
      call object%it_list%write (u)
    end if
    call object%prclib_stack%write (u)
  end subroutine

```



```

!    call write_separator_double (u)
    call beam_data_write (object%beam_data, u)
    call write_separator_double (u)
    call object%beam_structure%write (u)
    call write_separator_double (u)
    call object%pn%write (u)
    if (allocated (object%sample_fmt)) then
        call write_separator (u)
        write (u, "(1x,A)", advance="no") "Event sample formats = "
        do i = 1, size (object%sample_fmt)
            if (i > 1) write (u, "(A,1x)", advance="no") ", "
            write (u, "(A)", advance="no") char (object%sample_fmt(i))
        end do
        write (u, "(A)")
    end if
!    call write_separator_double (u)
    call object%process_stack%write (u)
end subroutine rt_data_write

```

Write only selected variables.

```

<RT data: rt data: TBP>+≡
    procedure :: write_vars => rt_data_write_vars

<RT data: procedures>+≡
    subroutine rt_data_write_vars (object, unit, vars)
        class(rt_data_t), intent(in) :: object
        integer, intent(in), optional :: unit
        type(string_t), dimension(:), intent(in), optional :: vars
        integer :: u, i
        u = output_unit (unit)
        if (present (vars)) then
            do i = 1, size (vars)
                call var_list_write_var (object%var_list, vars(i), unit = u, &
                    follow_link = .true.)
            end do
        end if
    end subroutine rt_data_write_vars

```

Write only the library stack.

```

<RT data: rt data: TBP>+≡
    procedure :: write_libraries => rt_data_write_libraries

<RT data: procedures>+≡
    subroutine rt_data_write_libraries (object, unit)
        class(rt_data_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit)
        call object%prclib_stack%write (u)
    end subroutine rt_data_write_libraries

```

Write only the beam data.

```

<RT data: rt data: TBP>+≡
    procedure :: write_beams => rt_data_write_beams

```

```

<RT data: procedures>+≡
subroutine rt_data_write_beams (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  call write_separator_double (u)
  call beam_data_write (object%beam_data, u)
  call write_separator_double (u)
  call object%beam_structure%write (u)
  call write_separator_double (u)
end subroutine rt_data_write_beams

```

Write only the process stack.

```

<RT data: rt data: TBP>+≡
  procedure :: write_process_stack => rt_data_write_process_stack

<RT data: procedures>+≡
subroutine rt_data_write_process_stack (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call object%process_stack%write (unit)
end subroutine rt_data_write_process_stack

```

## 19.5.4 Initialization

Initialize runtime data. This defines special variables such as `sqrts`, and should be done only for the instance that is actually global. Local copies will inherit the special variables.

```

<RT data: rt data: TBP>+≡
  procedure :: global_init => rt_data_global_init

<RT data: procedures>+≡
subroutine rt_data_global_init (global, paths)
  class(rt_data_t), intent(out), target :: global
  type(paths_t), intent(in), optional :: paths
  logical, target, save :: known = .true.
  real(default), parameter :: real_specimen = 1.
  call os_data_init (global%os_data, paths)
!   allocate (global%out_files)
!   allocate (global%rng)
!   call system_clock (global%seed)
!   call tao_random_create (global%rng, global%seed)
  call var_list_append_log_ptr &
    (global%var_list, var_str ("?logging"), logging, known, &
     intrinsic=.true.)
!   call var_list_append_int_ptr &
!     (global%var_list, var_str ("seed_value"), global%seed, known, &
!     intrinsic=.true.)
  call var_list_append_string &
    (global%var_list, var_str ("$_model_name"), &
     intrinsic=.true.)

```

```

call var_list_append_string &
  (global%var_list, var_str ("method"), var_str ("omega"), &
  intrinsic=.true.)
call var_list_append_log &
  (global%var_list, var_str ("report_progress"), .true., &
  intrinsic=.true.)
call var_list_append_string &
  (global%var_list, var_str ("restrictions"), var_str (""), &
  intrinsic=.true.)
call var_list_append_string &
  (global%var_list, var_str ("omega_flags"), var_str (""), &
  intrinsic=.true.)
!   call var_list_append_log &
!     (global%var_list, var_str ("read_color_factors"), .true., &
!     intrinsic=.true.)
!   call var_list_append_string &
!     (global%var_list, var_str ("user_procs_cut"), var_str (""), &
!     intrinsic=.true.)
!   call var_list_append_string &
!     (global%var_list, var_str ("user_procs_event_shape"), var_str (""), &
!     intrinsic=.true.)
!   call var_list_append_string &
!     (global%var_list, var_str ("user_procs_obs1"), var_str (""), &
!     intrinsic=.true.)
!   call var_list_append_string &
!     (global%var_list, var_str ("user_procs_obs2"), var_str (""), &
!     intrinsic=.true.)
!   call var_list_append_string &
!     (global%var_list, var_str ("user_procs_sf"), var_str (""), &
!     intrinsic=.true.)
!   call var_list_append_log &
!     (global%var_list, var_str ("slha_read_input"), .true., &
!     intrinsic=.true.)
!   call var_list_append_log &
!     (global%var_list, var_str ("slha_read_spectrum"), .true., &
!     intrinsic=.true.)
!   call var_list_append_log &
!     (global%var_list, var_str ("slha_read_decays"), .false., &
!     intrinsic=.true.)
call var_list_append_string &
  (global%var_list, var_str ("library_name"), &
  intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("sqrt_s"), &
  intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("beam1_momentum"), &
  intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("beam2_momentum"), &
  intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("crossing_angle"), &
  intrinsic=.true.)

```

```

call var_list_append_real &
  (global%var_list, var_str ("beams_theta"), &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("beams_phi"), &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("luminosity"), 0._default, &
   intrinsic=.true.)
if (present (paths)) then
  call var_list_append_string &
    (global%var_list, var_str ("lhpdf_dir"), paths%lhpdfdir, &
     intrinsic=.true.)
else
  call var_list_append_string &
    (global%var_list, var_str ("lhpdf_dir"), var_str(""), &
     intrinsic=.true.)
end if
call var_list_append_string &
  (global%var_list, var_str ("lhpdf_file"), var_str (""), &
   intrinsic=.true.)
call var_list_append_int &
  (global%var_list, var_str ("lhpdf_member"), 0, &
   intrinsic=.true.)
call var_list_append_int &
  (global%var_list, var_str ("lhpdf_photon_scheme"), 0, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("isr_alpha"), 0._default, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("isr_q_max"), 0._default, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("isr_mass"), 0._default, &
   intrinsic=.true.)
call var_list_append_int &
  (global%var_list, var_str ("isr_order"), 3, &
   intrinsic=.true.)
call var_list_append_log &
  (global%var_list, var_str ("?isr_recoil"), .false., &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("epa_alpha"), 0._default, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("epa_x_min"), 0._default, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("epa_q_min"), 0._default, &
   intrinsic=.true.)
call var_list_append_real &
  (global%var_list, var_str ("epa_e_max"), 0._default, &
   intrinsic=.true.)

```

```

call var_list_append_real &
    (global%var_list, var_str ("epa_mass"), 0._default, &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?epa_recoil"), .false., &
    intrinsic=.true.)
call var_list_append_real &
    (global%var_list, var_str ("ewa_x_min"), 0._default, &
    intrinsic=.true.)
call var_list_append_real &
    (global%var_list, var_str ("ewa_q_min"), 0._default, &
    intrinsic=.true.)
call var_list_append_real &
    (global%var_list, var_str ("ewa_pt_max"), 0._default, &
    intrinsic=.true.)
call var_list_append_real &
    (global%var_list, var_str ("ewa_mass"), 0._default, &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ewa_keep_momentum"), .false., &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ewa_keep_energy"), .false., &
    intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?circe1_photon1"), .false., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?circe1_photon2"), .false., &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("circe1_sqrts"), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?circe1_generate"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?circe1_map"), .true., &
!       intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("circe1_ver"), 0, intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("circe1_rev"), 0, intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("circe1_acc"), 1, intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("circe1_chat"), 0, intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("circe2_sqrts"), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?circe2_generate"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &

```

```

!      (global%var_list, var_str ("?circe2_map"), .true., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?circe2_polarized"), .true., &
!      intrinsic=.true.)
!      call var_list_append_string &
!      (global%var_list, var_str ("$circe2_file"), &
!      intrinsic=.true.)
!      call var_list_append_string &
!      (global%var_list, var_str ("$circe2_design"), var_str ("*"), &
!      intrinsic=.true.)
!      call var_list_append_string &
!      (global%var_list, var_str ("$beam_events_file"), &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?beam_events_warn_eof"), .true., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?alpha_s_is_fixed"), .true., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?alpha_s_from_lhapdf"), .false., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?alpha_s_from_pdf_builtin"), .false., &
!      intrinsic=.true.)
!      call var_list_append_int &
!      (global%var_list, var_str ("alpha_s_order"), 0, &
!      intrinsic=.true.)
!      call var_list_append_int &
!      (global%var_list, var_str ("alpha_s_nf"), 5, &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?alpha_s_from_mz"), .true., &
!      intrinsic=.true.)
!      call var_list_append_real &
!      (global%var_list, var_str ("lambda_qcd"), 200.e-3_default, &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?fatal_beam_decay"), .true., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?helicity_selection_active"), .true., &
!      intrinsic=.true.)
!      call var_list_append_real &
!      (global%var_list, var_str ("helicity_selection_threshold"), &
!      1E10_default, &
!      intrinsic=.true.)
!      call var_list_append_int &
!      (global%var_list, var_str ("helicity_selection_cutoff"), 1000, &
!      intrinsic=.true.)
!      call var_list_append_string &
!      (global%var_list, var_str ("$rng_method"), var_str ("tao"), &
!      intrinsic=.true.)

```

```

call var_list_append_string &
    (global%var_list, var_str ("integration_method"), var_str ("vamp"), &
    intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("threshold_calls"), 10, &
    intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("min_calls_per_channel"), 10, &
    intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("min_calls_per_bin"), 10, &
    intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("min_bins"), 3, &
    intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("max_bins"), 20, &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?stratified"), .true., &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?use_vamp_equivalences"), .true., &
    intrinsic=.true.)
call var_list_append_real &
    (global%var_list, var_str ("channel_weights_power"), 0.25_default, &
    intrinsic=.true.)
call var_list_append_string &
    (global%var_list, var_str ("phs_method"), var_str ("default"), &
    intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?vis_channels"), .false., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?check_phs_file"), .true., &
!       intrinsic=.true.)
call var_list_append_string &
    (global%var_list, var_str ("phs_file"), var_str (""), &
    intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?phs_only"), .false., &
    intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("phs_threshold_s"), 50._default, &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("phs_threshold_t"), 100._default, &
!       intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("phs_off_shell"), 2, &
!       intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("phs_t_channel"), 6, &
!       intrinsic=.true.)

```

```

!      call var_list_append_real &
!          (global%var_list, var_str ("phs_e_scale"), 10._default, &
!              intrinsic=.true.)
!      call var_list_append_real &
!          (global%var_list, var_str ("phs_m_scale"), 10._default, &
!              intrinsic=.true.)
!      call var_list_append_real &
!          (global%var_list, var_str ("phs_q_scale"), 10._default, &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?phs_keep_nonresonant"), .true., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?phs_step_mapping"), .true., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?phs_step_mapping_exp"), .false., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?phs_s_mapping"), .false., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?strfun_multichannel"), .false., &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$_run_id"), var_str (""), &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?use_best_grid"), .true., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?check_grid_file"), .true., &
!              intrinsic=.true.)
!      call var_list_append_real &
!          (global%var_list, var_str ("accuracy_goal"), 0._default, &
!              intrinsic=.true.)
!      call var_list_append_real &
!          (global%var_list, var_str ("error_goal"), 0._default, &
!              intrinsic=.true.)
!      call var_list_append_real &
!          (global%var_list, var_str ("relative_error_goal"), 0._default, &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?vis_history"), .false., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?isotropic_decay"), .false., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?diagonal_decay"), .false., &
!              intrinsic=.true.)
!      call var_list_append_log &
!          (global%var_list, var_str ("?check_event_file"), .true., &
!              intrinsic=.true.)

```



```

!     call var_list_append_string &
!         (global%var_list, var_str ("sevent_file_version"), var_str (""), &
!             intrinsic=.true.)
call var_list_append_int &
    (global%var_list, var_str ("n_events"), 0, &
        intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("allow_decays"), .true., &
!             intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("unweighted"), .true., &
        intrinsic=.true.)
!     call var_list_append_string &
!         (global%var_list, var_str ("sevent_normalization"), var_str ("auto"),&
!             intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("negative_weights"), .false., &
!             intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("use_num_id"), .false., &
!             intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("keep_beams"), .false., &
        intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("update_parameters"), .true., &
!             intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("update_scale"), .false., &
!             intrinsic=.true.)
!     call var_list_append_log &
!         (global%var_list, var_str ("update_alpha_s"), .false., &
!             intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("update_event"), .false., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("update_sqme"), .false., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("update_weight"), .false., &
        intrinsic=.true.)
call var_list_append_string &
    (global%var_list, var_str ("sample"), var_str (""), &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("read_raw"), .true., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("write_raw"), .true., &
        intrinsic=.true.)
!     call var_list_append_string &
!         (global%var_list, var_str ("extension_raw"), var_str ("evx"), &
!             intrinsic=.true.)

```

```

!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_default"), var_str ("evt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_debug"), var_str ("debug"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_hepevt"), var_str ("hepevt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_ascii_short"), var_str ("short.evt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_ascii_long"), var_str ("long.evt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_athena"), var_str ("athena.evt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_mokka"), var_str ("mokka.evt"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_lhef"), var_str ("lhef"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_lha"), var_str ("lha"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_hePMC"), var_str ("hePMC"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_stdhep"), var_str ("stdhep"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_stdhep_up"), var_str ("up.stdhep"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_hepevt_verbose"), var_str ("hepevt.verb"), &
!              intrinsic=.true.)
!      call var_list_append_string &
!          (global%var_list, var_str ("$extension_lha_verbose"), var_str ("lha.verb"), &
!              intrinsic=.true.)
!      call var_list_append_int (global%var_list, &
!          var_str ("n_bins"), 20, &
!              intrinsic=.true.)
!      call var_list_append_log (global%var_list, &
!          var_str ("?normalize_bins"), .false., &
!              intrinsic=.true.)
!      call var_list_append_string (global%var_list, &
!          var_str ("$obs_label"), var_str (""), &
!              intrinsic=.true.)
!      call var_list_append_string (global%var_list, &
!          var_str ("$obs_unit"), var_str (""), &
!              intrinsic=.true.)

```

```

!   call var_list_append_string (global%var_list, &
!       var_str ("title"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_string (global%var_list, &
!       var_str ("description"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_string (global%var_list, &
!       var_str ("x_label"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_string (global%var_list, &
!       var_str ("y_label"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("graph_width_mm"), 130, &
!       intrinsic=.true.)
!   call var_list_append_int &
!       (global%var_list, var_str ("graph_height_mm"), 90, &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?y_log"), .false., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?x_log"), .false., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?y_log"), .false., &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("x_min"), &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("x_max"), &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("y_min"), &
!       intrinsic=.true.)
!   call var_list_append_real &
!       (global%var_list, var_str ("y_max"), &
!       intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("gmlcode_bg"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("gmlcode_fg"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_histogram"), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_base"), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_pieewise"), &
!       intrinsic=.true.)

```

```

!   call var_list_append_log &
!       (global%var_list, var_str ("?fill_curve"), &
!           intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_curve"), &
!           intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_errors"), &
!           intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?draw_symbols"), &
!           intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$fill_options"), &
!           intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$draw_options"), &
!           intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$err_options"), &
!           intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$symbol"), &
!           intrinsic=.true.)
!   call var_list_append_real (global%var_list, &
!       var_str ("tolerance"), 0._default, &
!       intrinsic=.true.)
!   call var_list_append_int (global%var_list, &
!       var_str ("checkpoint"), intrinsic = .true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$out_file"), var_str (""), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?out_advance"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?out_custom"), .false., &
!       intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$out_comment"), var_str ("# "), &
!       intrinsic=.true.)
!   call var_list_append_string &
!       (global%var_list, var_str ("$out_separator"), var_str (" "), &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?out_columns"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?out_header"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &
!       (global%var_list, var_str ("?out_yerr"), .true., &
!       intrinsic=.true.)
!   call var_list_append_log &

```

```

!      (global%var_list, var_str ("?out_xerr"), .true., &
!      intrinsic=.true.)
! call var_list_append_int (global%var_list, var_str ("real_range"), &
!      range (real_specimen), intrinsic = .true., locked = .true.)
! call var_list_append_int (global%var_list, var_str ("real_precision"), &
!      precision (real_specimen), intrinsic = .true., locked = .true.)
! call var_list_append_real (global%var_list, var_str ("real_epsilon"), &
!      epsilon (real_specimen), intrinsic = .true., locked = .true.)
! call var_list_append_real (global%var_list, var_str ("real_tiny"), &
!      tiny (real_specimen), intrinsic = .true., locked = .true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?polarized_events"), .false., &
!      intrinsic=.true.)
! ! default settings for shower
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_fsr_active"), .false., &
!      intrinsic=.true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_use_PYTHIA_shower"), .true., &
!      intrinsic=.true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_PYTHIA_verbose"), .false., &
!      intrinsic=.true.)
! call var_list_append_string &
!      (global%var_list, var_str ("ps_PYTHIA_PYGIVE"), var_str (""), &
!      intrinsic=.true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_isr_active"), .false., &
!      intrinsic=.true.)
! call var_list_append_real (global%var_list, var_str ("ps_mass_cutoff"), &
!      1._default, intrinsic = .true.)
! call var_list_append_real (global%var_list, var_str ("ps_fsr_lambda"), &
!      0.29_default, intrinsic = .true.)
! call var_list_append_real (global%var_list, var_str ("ps_isr_lambda"), &
!      0.29_default, intrinsic = .true.)
! call var_list_append_int (global%var_list, var_str ("ps_max_n_flavors"), &
!      5, intrinsic = .true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_isr_alpha_s_running"), .true., &
!      intrinsic=.true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_fsr_alpha_s_running"), .true., &
!      intrinsic=.true.)
! call var_list_append_real (global%var_list, var_str ("ps_fixed_alpha_s"), &
!      0._default, intrinsic = .true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_isr_pt_ordered"), .false., &
!      intrinsic=.true.)
! call var_list_append_log &
!      (global%var_list, var_str ("?ps_isr_angular_ordered"), .true., &
!      intrinsic=.true.)
! call var_list_append_real (global%var_list, var_str ("ps_isr_primordial_kt_width"), &
!      0._default, intrinsic = .true.)
! call var_list_append_real (global%var_list, var_str ("ps_isr_primordial_kt_cutoff"), &

```

```

!      5._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("ps_isr_z_cutoff"), &
!      0.999_default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("ps_isr_minenergy"), &
!      1._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("ps_isr_tscalefactor"), &
!      1._default, intrinsic = .true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?ps_isr_only_onshell_emitted_partons"), .false., &
!      intrinsic=.true.)
!      ! default settings for hadronization
!      call var_list_append_log &
!      (global%var_list, var_str ("?hadronization_active"), .false., &
!      intrinsic=.true.)
!      ! setting for mlm matching
!      call var_list_append_log &
!      (global%var_list, var_str ("?mlm_matching"), .false., &
!      intrinsic=.true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Qcut_ME"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Qcut_PS"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_ptmin"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_etamax"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Rmin"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Emin"), &
!      0._default, intrinsic = .true.)
!      call var_list_append_int (global%var_list, var_str ("mlm_nmaxMEjets"), &
!      0, intrinsic = .true.)
!
!      call var_list_append_real (global%var_list, var_str ("mlm_ETclusfactor"), &
!      0.2_default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_ETclusminE"), &
!      5._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_etaclusfactor"), &
!      1._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Rclusfactor"), &
!      1._default, intrinsic = .true.)
!      call var_list_append_real (global%var_list, var_str ("mlm_Eclusfactor"), &
!      1._default, intrinsic = .true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?ckkw_matching"), .false., &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?muli_active"), .false., &
!      intrinsic=.true.)
!
!      call var_list_append_string (global%var_list, var_str ("$_datafile"), &
!      intrinsic=.true.)
!      call var_list_append_string (global%var_list, &
!      var_str ("$_comment_prefix"), var_str ("#"), intrinsic=.true.)

```

```

!      call var_list_append_log (global%var_list, var_str ("?write_header"), &
!      .true., intrinsic=.true.)
call var_list_append_string &
  (global%var_list, var_str ("pdf_builtin_set"), var_str ("CTEQ6L"), &
  intrinsic=.true.)
call var_list_append_log &
  (global%var_list, var_str ("?omega_omp"), &
  omp_is_active (), &
  intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?omp_is_active"), &
!      omp_is_active (), &
!      locked=.true., intrinsic=.true.)
!      call var_list_append_int &
!      (global%var_list, var_str ("omp_num_threads_default"), &
!      omp_get_default_max_threads (), &
!      locked=.true., intrinsic=.true.)
!      call var_list_append_int &
!      (global%var_list, var_str ("omp_num_threads"), &
!      omp_get_max_threads (), &
!      intrinsic=.true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("?dipole_resolve_emission"), &
!      intrinsic = .true.)
!      call var_list_append_string &
!      (global%var_list, var_str ("recombination_mode"), intrinsic = .true.)
!      call var_list_append_log &
!      (global%var_list, var_str ("recombination_complement"), &
!      intrinsic = .true.)
!      call var_list_append_real &
!      (global%var_list, var_str ("recombination_scale"), intrinsic = .true.)
!      call var_list_append_real &
!      (global%var_list, var_str ("photon_beam_separation"), &
!      intrinsic = .true.)
call global%init_pointer_variables ()
!      call iterations_lists_init_default (global%it_list_default)
end subroutine rt_data_global_init ! $

```

### 19.5.5 Local copies

This is done at compile time when a local copy of runtime data is needed: Link the variable list and initialize all derived parameters. This allows for synchronizing them with local variable changes without affecting global data.

Also re-initialize pointer variables, so they point to local copies of their targets.

```

<RT data: rt data: TBP>+≡
  procedure :: local_init => rt_data_local_init

<RT data: procedures>+≡
  subroutine rt_data_local_init (local, global, env)
    class(rt_data_t), intent(inout), target :: local
    type(rt_data_t), intent(in), target :: global

```

```

integer, intent(in), optional :: env
call var_list_link (local%var_list, global%var_list)
if (associated (global%model)) then
    call var_list_init_copies (local%var_list, &
        model_get_var_list_ptr (global%model), &
        derived_only = .true.)
end if
call local%init_pointer_variables ()
! if (present (env)) local%environment = env
end subroutine rt_data_local_init

```

These variables point to objects which get local copies:

```

<RT data: rt data: TBP>+≡
    procedure :: init_pointer_variables => rt_data_init_pointer_variables

<RT data: procedures>+≡
    subroutine rt_data_init_pointer_variables (local)
        class(rt_data_t), intent(inout), target :: local
        logical, target, save :: known = .true.
        call var_list_append_string_ptr &
            (local%var_list, var_str ("fc"), local%os_data%fc, known, &
            intrinsic=.true.)
        call var_list_append_string_ptr &
            (local%var_list, var_str ("fcflags"), local%os_data%fcflags, known, &
            intrinsic=.true.)
    end subroutine rt_data_init_pointer_variables

```

This should be executed when the compilation of a command that uses local variables is completed.

(Currently unused, keeping it for some time – just in case.)

```

<XXX RT data: public>≡
    public :: rt_data_local_reset

<XXX RT data: procedures>≡
    subroutine rt_data_local_reset (local)
        type(rt_data_t), intent(inout), target :: local
    end subroutine rt_data_local_reset

```

This is done at execution time: Copy data, transfer pointers. `local` has `intent(inout)` because its local variable list has already been prepared by the previous routine.

The process library and process stacks behave as global objects. The copies of the process library and process stacks should be shallow copies, so the contents stay identical. Since objects may be pushed on the stack in the local environment, upon restoring the global environment, we should reverse the assignment. Then the added stack elements will end up on the global stack. (This should be reconsidered in a parallel environment.)

```

<RT data: rt data: TBP>+≡
    procedure :: link => rt_data_link

```



```

<RT data: procedures>+≡
subroutine rt_data_link (local, global)
  class(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(in), target :: global
  local%lexer => global%lexer
  call var_list_link (local%var_list, global%var_list)
  if (associated (global%model)) then
    call var_list_synchronize (local%var_list, &
      model_get_var_list_ptr (global%model), reset_pointers = .true.)
  end if
  local%it_list = global%it_list
!   local%it_list_default => global%it_list_default
  local%os_data = global%os_data
  local%model => global%model
  local%prclib_stack = global%prclib_stack
  local%beam_data = global%beam_data
  local%beam_structure = global%beam_structure
  local%process_stack = global%process_stack
!   local%lhpdf_status = global%lhpdf_status
!   local%sf_list_allocated = .false.
!   local%sf_list => global%sf_list
  local%pn = global%pn
  if (allocated (local%sample_fmt)) deallocate (local%sample_fmt)
  if (allocated (global%sample_fmt)) then
    allocate (local%sample_fmt (size (global%sample_fmt)), &
      source = global%sample_fmt)
  end if
!   local%out_files => global%out_files
!   local%rng => global%rng
!   local%beam_polarization => global%beam_polarization
!   local%analysis_data_unit = global%analysis_data_unit
!   call nlo_setup_list_init (local%nlo_setup_list)
end subroutine rt_data_link

```

Restore the previous state of data; in particular, the variable list. This applies only to model variables (which are copies); other variables are automatically restored when local variables are removed.

Some command (`read_slha`) reads in model variables as local entities. In this case, the original values of model variables should not be restored, but the global variable list should be synchronized. However, this matters only if the local model is identical to the global model; otherwise, restoring will apply to a different model.

```

<RT data: rt data: TBP>+≡
  procedure :: restore => rt_data_restore

<RT data: procedures>+≡
subroutine rt_data_restore (global, local, keep_model_vars)
  class(rt_data_t), intent(inout) :: global
  type(rt_data_t), intent(inout) :: local
  logical, intent(in), optional :: keep_model_vars
  logical :: same_model, restore
  if (associated (global%model)) then
    same_model = &

```

```

        model_get_name (global%model) == model_get_name (local%model)
    if (present (keep_model_vars) .and. same_model) then
        restore = .not. keep_model_vars
    else
        if (.not. same_model) call msg_message ("Restoring model '" // &
            char (model_get_name (global%model)) // "'")
        restore = .true.
    end if
    if (restore) then
        call var_list_restore (global%var_list)
    else
        call var_list_synchronize &
            (global%var_list, model_get_var_list_ptr (global%model))
    end if
    call var_list_undefine (local%var_list, follow_link=.false.)
    global%prclib_stack = local%prclib_stack
    global%process_stack = local%process_stack
!    call nlo_setup_list_final (local%nlo_setup_list)
end subroutine rt_data_restore

```

### 19.5.6 Finalization

Finalizer for the variable list and the structure-function list. This is done only for the global RT dataset; local copies contain pointers to this and do not need a finalizer.

```

<RT data: rt data: TBP>+≡
    procedure :: final => rt_data_global_final

<RT data: procedures>+≡
    subroutine rt_data_global_final (global)
        class(rt_data_t), intent(inout) :: global
        call global%process_stack%final ()
        call global%prclib_stack%final ()
        call var_list_final (global%var_list)
!        if (global%sf_list_allocated) then
!            call sf_list_final (global%sf_list)
!            deallocate (global%sf_list)
!            global%sf_list_allocated = .false.
!        end if
!        deallocate (global%it_list_default)
!        call file_list_final (global%out_files)
!        deallocate (global%out_files)
!        deallocate (global%rng)
    end subroutine rt_data_global_final

```

### 19.5.7 Filling Contents

Add a library (available via a pointer of type `prclib_entry_t`) to the stack and update the pointer and variable list to the current library. The pointer

association of `prclib_entry` will be discarded.

```

<RT data: rt data: TBP>+≡
  procedure :: add_prclib => rt_data_add_prclib

<RT data: procedures>+≡
  subroutine rt_data_add_prclib (global, prclib_entry)
    class(rt_data_t), intent(inout) :: global
    type(prclib_entry_t), intent(inout), pointer :: prclib_entry
    call global%prclib_stack%push (prclib_entry)
    call global%update_prclib (global%prclib_stack%get_first_ptr ())
  end subroutine rt_data_add_prclib

```

Given a pointer to a process library, make this the currently active library.

```

<RT data: rt data: TBP>+≡
  procedure :: update_prclib => rt_data_update_prclib

<RT data: procedures>+≡
  subroutine rt_data_update_prclib (global, lib)
    class(rt_data_t), intent(inout) :: global
    type(process_library_t), intent(in), target :: lib
    type(var_entry_t), pointer :: var
    global%prclib => lib
    var => var_list_get_var_ptr (global%var_list, &
      var_str ("$_library_name"), follow_link = .false.)
    if (associated (var)) then
      call var_entry_set_string (var, &
        global%prclib%get_name (), is_known=.true.)
    else
      call var_list_append_string (global%var_list, &
        var_str ("$_library_name"), global%prclib%get_name (), &
        intrinsic = .true.)
    end if
  end subroutine rt_data_update_prclib

```

## 19.5.8 Get contents

The helicity selection data are distributed among several parameters. Here, we collect them in a single record.

```

<RT data: rt data: TBP>+≡
  procedure :: get_helicity_selection => rt_data_get_helicity_selection

<RT data: procedures>+≡
  function rt_data_get_helicity_selection (rt_data) result (helicity_selection)
    class(rt_data_t), intent(in) :: rt_data
    type(helicity_selection_t) :: helicity_selection
    associate (var_list => rt_data%var_list)
      helicity_selection%active = var_list_get_lval (var_list, &
        var_str ("?helicity_selection_active"))
      if (helicity_selection%active) then
        helicity_selection%threshold = var_list_get_rval (var_list, &
          var_str ("helicity_selection_threshold"))
        helicity_selection%cutoff = var_list_get_ival (var_list, &
          var_str ("helicity_selection_cutoff"))
      end if
    end associate
  end function rt_data_get_helicity_selection

```

```

        end if
    end associate
end function rt_data_get_helicity_selection

```

### 19.5.9 Auxiliary stuff

Write a separator line.

```

<RT data: procedures>+≡
subroutine write_separator (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("-", 72)
end subroutine write_separator

subroutine write_separator_double (u)
    integer, intent(in) :: u
    write (u, "(A)") repeat ("=", 72)
end subroutine write_separator_double

```

### 19.5.10 Test

This is the master for calling self-test procedures.

```

<RT data: public>+≡
public :: rt_data_test

<RT data: tests>≡
subroutine rt_data_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <RT data: execute tests>
end subroutine rt_data_test

```

### Initial content

Display the RT data in the state just after (global) initialization.

```

<RT data: execute tests>≡
call test (rt_data_1, "rt_data_1", &
    "initialize", &
    u, results)

<RT data: tests>+≡
subroutine rt_data_1 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: rt_data

    write (u, "(A)")  "* Test output: rt_data_1"
    write (u, "(A)")  "* Purpose: initialize global runtime data"
    write (u, "(A)")

    call rt_data%global_init ()
    call rt_data%it_list%init ([2, 3], [5000, 20000])

```

```

call var_list_set_log (rt_data%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)
rt_data%os_data%fc = "Fortran-compiler"
rt_data%os_data%fcflags = "Fortran-flags"
call rt_data%write (u)

call rt_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_1"

end subroutine rt_data_1

```

## Fill values

Fill in empty slots in the runtime data block.

```

<RT data: execute tests>+=
call test (rt_data_2, "rt_data_2", &
    "fill", &
    u, results)

<RT data: tests>+=
subroutine rt_data_2 (u)
integer, intent(in) :: u
type(rt_data_t), target :: rt_data
type(var_list_t), pointer :: model_vars
type(flavor_t), dimension(2) :: flv
type(string_t) :: cut_expr_text
type(ifile_t) :: ifile
type(stream_t) :: stream
type(parse_tree_t) :: parse_tree

write (u, "(A)")  "* Test output: rt_data_2"
write (u, "(A)")  "* Purpose: initialize global runtime data &
    &and fill contents"
write (u, "(A)")

call syntax_model_file_init ()

call rt_data%global_init ()

call model_list_read_model (var_str ("Test"), &
    var_str ("Test.mdl"), rt_data%os_data, rt_data%model)
call var_list_set_string (rt_data%var_list, var_str ("$model_name"), &
    model_get_name (rt_data%model), is_known=.true.)
model_vars => model_get_var_list_ptr (rt_data%model)
call var_list_init_copies &
    (rt_data%var_list, model_vars)
call var_list_synchronize &
    (rt_data%var_list, model_vars, reset_pointers = .true.)

call var_list_set_real (rt_data%var_list, var_str ("sqrts"),&
    1000._default, is_known = .true.)

```

```

call flavor_init (flv, [25,25], rt_data%model)
call beam_data_init_sqrts (rt_data%beam_data, &
    var_list_get_rval (rt_data%var_list, var_str ("sqrts")), flv)

call var_list_set_string (rt_data%var_list, var_str ("run_id"), &
    var_str ("run1"), is_known = .true.)
call var_list_set_real (rt_data%var_list, var_str ("luminosity"), &
    33._default, is_known = .true.)

call syntax_pexpr_init ()
cut_expr_text = "all Pt > 100 [s]"
call ifile_append (ifile, cut_expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)
rt_data%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

allocate (rt_data%sample_fmt (2))
rt_data%sample_fmt(1) = "foo_fmt"
rt_data%sample_fmt(2) = "bar_fmt"

call var_list_set_log (rt_data%var_list, var_str ("?omega_omp"), &
    .false., is_known = .true.)
rt_data%os_data%fc = "Fortran-compiler"
rt_data%os_data%fcflags = "Fortran-flags"

call rt_data%write (u)

call parse_tree_final (parse_tree)
call stream_final (stream)
call ifile_final (ifile)
call syntax_pexpr_final ()

call rt_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_2"

end subroutine rt_data_2

```

## Save and restore

Set up a local runtime data block, change some contents, restore the global block.

For a model variable, the save-restore procedure is nontrivial. We first allocate a local copy of this variable and set a pointer to the original model variable. We set a new value, update the dependent variables and synchronize them with the local variable list, if modified. When restoring the old data, we recover the model variables from the stored values and delete the local instances.

```

<RT data: execute tests>+=
call test (rt_data_3, "rt_data_3", &
    "save/restore", &

```

```

        u, results)

<RT data: tests>+≡
subroutine rt_data_3 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: rt_data, local
    type(var_list_t), pointer :: model_vars
    type(var_entry_t), pointer :: var
    type(flavor_t), dimension(2) :: flv
    type(string_t) :: cut_expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: parse_tree
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: rt_data_3"
    write (u, "(A)")  "* Purpose: initialize global runtime data &
        &and fill contents;"
    write (u, "(A)")  "* copy to local block and back"
    write (u, "(A)")

    write (u, "(A)")  "* Init global data"
    write (u, "(A)")

    call syntax_model_file_init ()

    call rt_data%global_init ()

    call model_list_read_model (var_str ("Test"), &
        var_str ("Test.mdl"), rt_data%os_data, rt_data%model)
    call var_list_set_string (rt_data%var_list, var_str ("model_name"), &
        model_get_name (rt_data%model), is_known=.true.)
    model_vars => model_get_var_list_ptr (rt_data%model)
    call var_list_init_copies &
        (rt_data%var_list, model_vars)
    call var_list_synchronize &
        (rt_data%var_list, model_vars, reset_pointers = .true.)

    call var_list_set_real (rt_data%var_list, var_str ("sqrts"), &
        1000._default, is_known = .true.)
    call flavor_init (flv, [25,25], rt_data%model)
    call beam_data_init_sqrts (rt_data%beam_data, &
        var_list_get_rval (rt_data%var_list, var_str ("sqrts")), flv)

    call rt_data%beam_structure%init ([1], rt_data%var_list)
    call rt_data%beam_structure%set_entry (1, 1, var_str ("pdf_builtin"))

    call var_list_set_string (rt_data%var_list, var_str ("run_id"), &
        var_str ("run1"), is_known = .true.)
    call var_list_set_real (rt_data%var_list, var_str ("luminosity"), &
        33._default, is_known = .true.)

    call syntax_pexpr_init ()
    cut_expr_text = "all Pt > 100 [s]"
    call ifile_append (ifile, cut_expr_text)

```

```

call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)
rt_data%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

allocate (rt_data%sample_fmt (2))
rt_data%sample_fmt(1) = "foo_fmt"
rt_data%sample_fmt(2) = "bar_fmt"

call var_list_set_log (rt_data%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)
rt_data%os_data%fc = "Fortran-compiler"
rt_data%os_data%fcflags = "Fortran-flags"

allocate (lib)
call lib%init (var_str ("library_1"))
call rt_data%add_prclib (lib)

write (u, "(A)")  "* Init and modify local data"
write (u, "(A)")

call local%local_init (rt_data)
call local%link (rt_data)

var => var_list_get_var_ptr (local%var_list, var_str ("ms"))
if (var_entry_is_copy (var)) then
    call var_list_init_copy (local%var_list, var, user=.true.)
    call var_list_set_original_pointer (local%var_list, var_str ("ms"), &
        model_vars)
    call var_list_set_real (local%var_list, var_str ("ms"), &
        150._default, is_known = .true., model_name = var_str ("Test"))
    call model_parameters_update (local%model)
    call var_list_synchronize (local%var_list, model_vars)
end if

call var_list_append_string (local%var_list, &
    var_str ("integration_method"), intrinsic = .true., user = .true.)
call var_list_set_string (local%var_list, var_str ("integration_method"), &
    var_str ("midpoint"), is_known = .true.)

call var_list_append_string (local%var_list, &
    var_str ("phs_method"), intrinsic = .true., user = .true.)
call var_list_set_string (local%var_list, var_str ("phs_method"), &
    var_str ("single"), is_known = .true.)

local%os_data%fc = "Local compiler"

allocate (lib)
call lib%init (var_str ("library_2"))
call local%add_prclib (lib)

call local%write (u)

write (u, "(A)")
write (u, "(A)")  "* Restore global data"

```



```

write (u, "(A)")

call rt_data%restore (local)
call rt_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call parse_tree_final (parse_tree)
call stream_final (stream)
call ifile_final (ifile)
call syntax_pexpr_final ()

call rt_data%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_3"

end subroutine rt_data_3

```

## Show variables

Display selected variables in the global record.

```

<RT data: execute tests>+≡
  call test (rt_data_4, "rt_data_4", &
    "show variables", &
    u, results)

<RT data: tests>+≡
  subroutine rt_data_4 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: rt_data

    type(string_t), dimension(0) :: empty_string_array

    write (u, "(A)")  "* Test output: rt_data_4"
    write (u, "(A)")  "* Purpose: display selected variables"
    write (u, "(A)")

    call rt_data%global_init ()

    write (u, "(A)")  "* No variables:"
    write (u, "(A)")

    call rt_data%write_vars (u, empty_string_array)

    write (u, "(A)")  "* Two variables:"
    write (u, "(A)")

    call rt_data%write_vars (u, &
      [var_str ("?unweighted"), var_str ("phs_method")])

```

```

write (u, "(A)")
write (u, "(A)")  "* Display whole record with selected variables"
write (u, "(A)")

call rt_data%write (u, &
    vars = [var_str ("?unweighted"), var_str ("phs_method")])

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_4"

end subroutine rt_data_4

```

## Show parts

Display only selected parts in the state just after (global) initialization.

```

<RT data: execute tests>+≡
call test (rt_data_5, "rt_data_5", &
    "show parts", &
    u, results)

<RT data: tests>+≡
subroutine rt_data_5 (u)
integer, intent(in) :: u
type(rt_data_t), target :: rt_data

write (u, "(A)")  "* Test output: rt_data_5"
write (u, "(A)")  "* Purpose: display parts of rt data"
write (u, "(A)")

call rt_data%global_init ()
call rt_data%write_libraries (u)

write (u, "(A)")

call rt_data%write_beams (u)

write (u, "(A)")

call rt_data%write_process_stack (u)

call rt_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_5"

end subroutine rt_data_5

```

## 19.6 Select implementations

For abstract types (process core, integrator, phase space, etc.), we need a way to dynamically select a concrete type, using either data given by the user or a

previous selection of a concrete type. This is done by subroutines in the current module.

This module provides no new types, just procedures.

*<dispatch.f90>*≡

*<File header>*

module dispatch

*<Use kinds>*

*<Use strings>*

use limits, only: MZ\_REF, ALPHA\_QCD\_MZ\_REF !NODEP!

use constants, only: PI !NODEP!

*<Use file utils>*

use diagnostics !NODEP!

use os\_interface

use unit\_tests

use sm\_qcd

use variables

use flavors

use interactions

use models

use process\_constants

use pdf\_builtin !NODEP!

use sf\_mappings

use sf\_base

use sf\_pdf\_builtin

use sf\_lhapdf

use sf\_circe1

use sf\_circe2

use sf\_lumi\_linker

use sf\_isr

use sf\_epa

use sf\_ewa

use sf\_escan

use sf\_beam\_events

use sf\_user

use rng\_base

use rng\_tao

use mci\_base

use mci\_midpoint

use mci\_vamp

use phs\_base

use mappings

use phs\_forests

use phs\_single

use phs\_wood

use prc\_core\_def

use prc\_test

use beams

use prc\_omega

use prc\_core

use processes

use beam\_structures

use eio\_base

```

use eio_raw
use eio_lhef
use eio_hePMC
use eio_weights
use rt_data

<Standard module head>

<Dispatch: public>

contains

<Dispatch: procedures>

<Dispatch: tests>

end module dispatch

```

### 19.6.1 Process Core Definition

The `prc_core_def_t` abstract type can be instantiated by providing a `$method` string variable.

Note: `core_def` has `intent(inout)` because gfortran 4.7.1 crashes for `intent(out)`.

```

<Dispatch: public>≡
    public :: dispatch_core_def

<Dispatch: procedures>≡
    subroutine dispatch_core_def (core_def, prt_in, prt_out, global)
        class(prc_core_def_t), allocatable, intent(inout) :: core_def
        type(string_t), dimension(:), intent(in) :: prt_in
        type(string_t), dimension(:), intent(in) :: prt_out
        type(rt_data_t), intent(in) :: global
        type(string_t) :: method
        type(string_t) :: model_name
        type(string_t) :: restrictions
        logical :: openmp_support
        logical :: report_progress
        type(string_t) :: extra_options
        associate (var_list => global%var_list)
            method = var_list_get_sval (var_list, var_str ("method"))
            model_name = var_list_get_sval (var_list, var_str ("model_name"))
            select case (char (method))
            case ("unit_test")
                allocate (prc_test_def_t :: core_def)
                select type (core_def)
                type is (prc_test_def_t)
                    call core_def%init (model_name, prt_in, prt_out)
                end select
            case ("omega")
                restrictions = var_list_get_sval (var_list, &
                    var_str ("restrictions"))
                openmp_support = var_list_get_lval (var_list, &
                    var_str ("?omega_openmp"))
            end select
        end associate
    end subroutine

```

```

        report_progress = var_list_get_lval (var_list, &
            var_str ("?report_progress"))
        extra_options = var_list_get_sval (var_list, &
            var_str ("$omega_flags"))
        allocate (omega_def_t :: core_def)
        select type (core_def)
        type is (omega_def_t)
            call core_def%init (model_name, prt_in, prt_out, &
                restrictions, openmp_support, report_progress, extra_options)
        end select
    case default
        call msg_fatal ("Process configuration: method '" &
            // char (method) // "' not implemented")
    end select
end associate
end subroutine dispatch_core_def

```

### 19.6.2 Process core allocation

Here we allocate an object of abstract type `prc_core_t` with a concrete type that matches a process definition. The `prc_omega_t` extension will require the current parameter set, so we take the opportunity to grab it from the model.

```

<Dispatch: public>+≡
    public :: dispatch_core

<Dispatch: procedures>+≡
    subroutine dispatch_core (core, core_def, model, helicity_selection, qcd)
        class(prc_core_t), allocatable, intent(inout) :: core
        class(prc_core_def_t), intent(in) :: core_def
        type(model_t), intent(in), target, optional :: model
        type(helicity_selection_t), intent(in), optional :: helicity_selection
        type(qcd_t), intent(in), optional :: qcd
        select type (core_def)
        type is (prc_test_def_t)
            allocate (test_t :: core)
        type is (omega_def_t)
            allocate (prc_omega_t :: core)
        select type (core)
        type is (prc_omega_t)
            call core%set_parameters (model, helicity_selection, qcd)
        end select
    class default
        call msg_bug ("Process core: unexpected process definition type")
    end select
end subroutine dispatch_core

```

### 19.6.3 Process core update and restoration

Here we take an existing object of abstract type `prc_core_t` and update the parameters as given by the current state of `model`. Optionally, we can save the

previous state as `saved_core`. The second routine restores the original from the save.

(In the test case, there is no possible update.)

```

<Dispatch: public>+≡
  public :: dispatch_core_update
  public :: dispatch_core_restore

<Dispatch: procedures>+≡
  subroutine dispatch_core_update (core, model, helicity_selection, qcd, &
    saved_core)
    class(prc_core_t), allocatable, intent(inout) :: core
    type(model_t), intent(in), optional, target :: model
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    type(qcd_t), intent(in), optional :: qcd
    class(prc_core_t), allocatable, intent(inout), optional :: saved_core
    if (present (saved_core)) then
      allocate (saved_core, source = core)
    end if
    select type (core)
    type is (test_t)
    type is (prc_omega_t)
      call core%set_parameters (model, helicity_selection, qcd)
      call core%activate_parameters ()
    class default
      call msg_bug ("Process core update: unexpected process definition type")
    end select
  end subroutine dispatch_core_update

  subroutine dispatch_core_restore (core, saved_core)
    class(prc_core_t), allocatable, intent(inout) :: core
    class(prc_core_t), allocatable, intent(inout) :: saved_core
    call move_alloc (from = saved_core, to = core)
    select type (core)
    type is (test_t)
    type is (prc_omega_t)
      call core%activate_parameters ()
    class default
      call msg_bug ("Process core restore: unexpected process definition type")
    end select
  end subroutine dispatch_core_restore

```

#### 19.6.4 Integrator allocation

Allocate an integrator according to the variable `$integration_method`.

```

<Dispatch: public>+≡
  public :: dispatch_mci

<Dispatch: procedures>+≡
  subroutine dispatch_mci (mci, global, process_id)
    class(mci_t), allocatable, intent(inout) :: mci
    type(rt_data_t), intent(in) :: global
    type(string_t), intent(in) :: process_id
    type(string_t) :: run_id

```

```

type(string_t) :: integration_method
type(grid_parameters_t) :: grid_par
logical :: rebuild_grids
integration_method = var_list_get_sval (global%var_list, &
    var_str ("integration_method"))
select case (char (integration_method))
case ("midpoint")
    allocate (mci_midpoint_t :: mci)
case ("vamp", "default")
    associate (var_list => global%var_list)
        grid_par%threshold_calls = &
            var_list_get_ival (var_list, var_str ("threshold_calls"))
        grid_par%min_calls_per_channel = &
            var_list_get_ival (var_list, var_str ("min_calls_per_channel"))
        grid_par%min_calls_per_bin = &
            var_list_get_ival (var_list, var_str ("min_calls_per_bin"))
        grid_par%min_bins = &
            var_list_get_ival (var_list, var_str ("min_bins"))
        grid_par%max_bins = &
            var_list_get_ival (var_list, var_str ("max_bins"))
        grid_par%stratified = &
            var_list_get_lval (var_list, var_str ("?stratified"))
        grid_par%use_vamp_equivalences = &
            var_list_get_lval (var_list, var_str ("?use_vamp_equivalences"))
        grid_par%channel_weights_power = &
            var_list_get_rval (var_list, var_str ("channel_weights_power"))
        run_id = &
            var_list_get_sval (var_list, var_str ("run_id"))
        rebuild_grids = &
            var_list_get_lval (var_list, var_str ("?rebuild_grids"))
    end associate
    allocate (mci_vamp_t :: mci)
    select type (mci)
    type is (mci_vamp_t)
        call mci%set_grid_parameters (grid_par)
        if (run_id /= "") then
            call mci%set_grid_filename (process_id, run_id)
        else
            call mci%set_grid_filename (process_id)
        end if
        call mci%set_rebuild_flag (rebuild_grids)
    end select
case default
    call msg_fatal ("Integrator '" &
        // char (integration_method) // "' not implemented")
end select
end subroutine dispatch_mci

```

### 19.6.5 Phase-space allocation

Allocate a phase-space object according to the variable \$phs\_method.

(Dispatch: public) +≡

```

public :: dispatch_phs
<Dispatch: procedures>+=
subroutine dispatch_phs (phs, global, mapping_defaults, phs_par)
  class(phs_config_t), allocatable, intent(inout) :: phs
  type(rt_data_t), intent(in) :: global
  type(mapping_defaults_t), intent(in), optional :: mapping_defaults
  type(phs_parameters_t), intent(in), optional :: phs_par
  type(string_t) :: phs_method, phs_file
  logical :: use_equivalences
  integer :: u_phs
  logical :: exist
  phs_method = var_list_get_sval (global%var_list, &
    var_str ("phs_method"))
  phs_file = var_list_get_sval (global%var_list, &
    var_str ("phs_file"))
  use_equivalences = var_list_get_lval (global%var_list, &
    var_str ("use_vamp_equivalences"))
  select case (char (phs_method))
  case ("single")
    allocate (phs_single_config_t :: phs)
  case ("wood", "default")
    allocate (phs_wood_config_t :: phs)
    select type (phs)
    type is (phs_wood_config_t)
      if (phs_file /= "") then
        inquire (file = char (phs_file), exist = exist)
        if (exist) then
          call msg_message ("Phase space: reading configuration from '" &
            // char (phs_file) // "'")
          u_phs = free_unit ()
          open (u_phs, file = char (phs_file), &
            action = "read", status = "old")
          call phs%set_input (u_phs)
        else
          call msg_fatal ("Phase space: configuration file '" &
            // char (phs_file) // "' not found")
        end if
      end if
    end if
    if (present (phs_par)) &
      call phs%set_parameters (phs_par)
    if (use_equivalences) &
      call phs%enable_equivalences ()
    if (present (mapping_defaults)) &
      call phs%set_mapping_defaults (mapping_defaults)
  end select
  case default
    call msg_fatal ("Phase space: parameterization method '" &
      // char (phs_method) // "' not implemented")
  end select
end subroutine dispatch_phs

```



## 19.6.6 Random-number generator

Allocate a random-number generator according to the variable `$rng_method`. Do not seed the generator, yet.

```
<Dispatch: public>+≡
    public :: dispatch_rng

<Dispatch: procedures>+≡
    subroutine dispatch_rng (rng, global)
        class(rng_t), allocatable, intent(inout) :: rng
        type(rt_data_t), intent(in) :: global
        type(string_t) :: rng_method
        rng_method = var_list_get_sval (global%var_list, &
            var_str ("rng_method"))
        select case (char (rng_method))
        case ("unit_test")
            allocate (rng_test_t :: rng)
        case ("tao")
            allocate (rng_tao_t :: rng)
        case default
            call msg_fatal ("Random-number generator '" &
                // char (rng_method) // "' not implemented")
        end select
    end subroutine dispatch_rng
```

## 19.6.7 Structure function configuration data

Allocate a structure-function configuration object according to the `sf_method` string.

The `global` object is `intent(inout)` because the PDF status may change during initialization.

```
<Dispatch: public>+≡
    public :: dispatch_sf_data

<Dispatch: procedures>+≡
    subroutine dispatch_sf_data (data, sf_method, i_beam, global)
        class(sf_data_t), allocatable, intent(inout) :: data
        type(string_t), intent(in) :: sf_method
        integer, dimension(:), intent(in) :: i_beam
        type(rt_data_t), intent(inout) :: global
        type(model_t), pointer :: model
        type(flavor_t), dimension(2) :: flv_beam
        real(default) :: isr_alpha, isr_q_max, isr_mass
        integer :: isr_order
        logical :: isr_recoil
        real(default) :: epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass
        logical :: epa_recoil
        real(default) :: ewa_x_min, ewa_q_min, ewa_pt_max, ewa_mass
        logical :: ewa_keep_momentum, ewa_keep_energy
        type(string_t) :: pdf_name, pdf_path
        flv_beam = beam_data_get_flavor (global%beam_data)
        model => global%model
        associate (var_list => global%var_list)
```

```

select case (char (sf_method))
case ("sf_test_0", "sf_test_1")
    allocate (sf_test_data_t :: data)
    select type (data)
    type is (sf_test_data_t)
        select case (char (sf_method))
        case ("sf_test_0"); call data%init (model, flv_beam(i_beam(1)))
        case ("sf_test_1"); call data%init (model, flv_beam(i_beam(1)), &
            mode = 1)
        end select
    end select
case ("pdf_builtin")
    allocate (pdf_builtin_data_t :: data)
    select type (data)
    type is (pdf_builtin_data_t)
        pdf_name = &
            var_list_get_sval (var_list, var_str ("pdf_builtin_set"))
        call data%init (global%pdf_builtin_status, &
            model, flv_beam(i_beam(1)), &
            name = pdf_name, &
            path = global%os_data%pdf_builtin_datapath)
    end select
case ("lhpdf")
    allocate (lhpdf_data_t :: data)
    select type (data)
    type is (lhpdf_data_t)
        call data%init &
            (global%lhpdf_status, model, flv_beam(i_beam(1)))
    end select
case ("isr")
    allocate (isr_data_t :: data)
    isr_alpha = var_list_get_rval (var_list, var_str ("isr_alpha"))
    if (isr_alpha == 0) then
        isr_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
            ** 2 / (4 * PI)
    end if
    isr_q_max = var_list_get_rval (var_list, var_str ("isr_q_max"))
    if (isr_q_max == 0) then
        isr_q_max = global%beam_data%sqrts
    end if
    isr_mass = var_list_get_rval (var_list, var_str ("isr_mass"))
    isr_order = var_list_get_ival (var_list, var_str ("isr_order"))
    isr_recoil = var_list_get_lval (var_list, var_str ("isr_recoil"))
    select type (data)
    type is (isr_data_t)
        call data%init &
            (model, flv_beam (i_beam(1)), isr_alpha, isr_q_max, &
            isr_mass, isr_order, isr_recoil)
        call data%check ()
    end select
case ("epa")
    allocate (epa_data_t :: data)
    epa_alpha = var_list_get_rval (var_list, var_str ("epa_alpha"))
    if (epa_alpha == 0) then

```

```

        epa_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
            ** 2 / (4 * PI)
    end if
    epa_x_min = var_list_get_rval (var_list, var_str ("epa_x_min"))
    epa_q_min = var_list_get_rval (var_list, var_str ("epa_q_min"))
    epa_e_max = var_list_get_rval (var_list, var_str ("epa_e_max"))
    if (epa_e_max == 0) then
        epa_e_max = global%beam_data%sqrts
    end if
    epa_mass = var_list_get_rval (var_list, var_str ("epa_mass"))
    epa_recoil = var_list_get_lval (var_list, var_str ("?epa_recoil"))
    select type (data)
    type is (epa_data_t)
        call data%init &
            (model, flv_beam (i_beam(1)), epa_alpha, epa_x_min, &
                epa_q_min, epa_e_max, epa_mass, epa_recoil)
        call data%check ()
    end select
case ("ewa")
    allocate (ewa_data_t :: data)
    ewa_x_min = var_list_get_rval (var_list, var_str ("ewa_x_min"))
    ewa_q_min = var_list_get_rval (var_list, var_str ("ewa_q_min"))
    ewa_pt_max = var_list_get_rval (var_list, var_str ("ewa_pt_max"))
    if (ewa_pt_max == 0) then
        ewa_pt_max = global%beam_data%sqrts
    end if
    ewa_mass = var_list_get_rval (var_list, var_str ("ewa_mass"))
    ewa_keep_momentum = var_list_get_lval (var_list, &
        var_str ("?ewa_keep_momentum"))
    ewa_keep_energy = var_list_get_lval (var_list, &
        var_str ("?ewa_keep_energy"))
    if (ewa_keep_momentum .and. ewa_keep_energy) &
        call msg_fatal (" EWA cannot conserve both energy " &
            // "and momentum.")
    select type (data)
    type is (ewa_data_t)
        if (ewa_mass /= 0) then
            call data%init &
                (model, flv_beam (i_beam(1)), ewa_x_min, ewa_q_min, &
                    ewa_pt_max, global%beam_data%sqrts, ewa_keep_momentum, &
                    ewa_keep_energy, ewa_mass)
        else
            call data%init &
                (model, flv_beam (i_beam(1)), ewa_x_min, ewa_q_min, &
                    ewa_pt_max, global%beam_data%sqrts, ewa_keep_momentum, &
                    ewa_keep_energy)
        end if
        call data%check ()
    end select
case ("energy_scan")
case default
    call msg_bug ("Structure function '" &
        // char (sf_method) // "' not implemented yet")
end select

```

```

        end associate
    end subroutine dispatch_sf_data

```

This is an auxiliary procedure, used by the beam-structure expansion: tell for a given structure function name, whether it corresponds to a pair spectrum ( $n = 2$ ), a single-particle structure function ( $n = 1$ ), or nothing ( $n = 0$ ).

*(Dispatch: procedures)*+≡

```

function strfun_mode (name) result (n)
    type(string_t), intent(in) :: name
    integer :: n
    select case (char (name))
    case ("none")
        n = 0
    case ("sf_test_0", "sf_test_1")
        n = 1
    case ("pdf_builtin", "lhpdf", "isr", "epa", "ewa")
        n = 1
    case default
        call msg_bug ("Structure function '" // char (name) &
            // "' not supported yet")
    end select
end function strfun_mode

```

Dispatch a whole structure-function chain, given beam data and beam structure data.

This could be done generically, but we should look at the specific combination of structure functions in order to select appropriate mappings.

The `beam_structure` argument gets copied because we want to expand it to canonical form (one valid structure-function entry per record) before proceeding further.

Note: Should add global mappings, as appropriate for a specific combination of structure functions.

*(Dispatch: public)*+≡

```

public :: dispatch_sf_config

```

*(Dispatch: procedures)*+≡

```

subroutine dispatch_sf_config (sf_config, sf_channel, global)
    type(sf_config_t), dimension(:), allocatable, intent(out) :: sf_config
    type(sf_channel_t), dimension(:), allocatable, intent(out) :: sf_channel
    type(rt_data_t), intent(inout) :: global
    type(beam_structure_t) :: beam_structure
    class(sf_data_t), allocatable :: sf_data
    integer :: n_record, i
    beam_structure = global%beam_structure
    call beam_structure%expand (strfun_mode)
    n_record = beam_structure%get_n_record ()
    allocate (sf_config (n_record))
    do i = 1, n_record
        call dispatch_sf_data (sf_data, &
            beam_structure%get_name (i), &
            beam_structure%get_i_entry (i), &
            global)
    end do
end subroutine dispatch_sf_config

```

```

        call sf_config(i)%init (beam_structure%get_i_entry (i), sf_data)
        deallocate (sf_data)
    end do
    select case (char (beam_structure%to_string ()))
    !!! JRR: WK please check
    !!! What about this? How for the other SFs?
    !!! Single structure function not covered yet
    case ("pdf_builtin, none => none, pdf_builtin")
        allocate (sf_channel (1))
        call sf_channel(1)%init (2)
        call sf_channel(1)%set_s_mapping ([1, 2], power = 2._default)
    case ("lhpdf, none => none, lhpdf")
        allocate (sf_channel (1))
        call sf_channel(1)%init (2)
        call sf_channel(1)%set_s_mapping ([1, 2], power = 2._default)
    case ("isr, none => none, isr")
        allocate (sf_channel (1))
        call sf_channel(1)%init(2)
    case ("epa, none => none, epa")
        allocate (sf_channel (1))
        call sf_channel(1)%init(2)
    end select
end subroutine dispatch_sf_config

```

### 19.6.8 Event I/O stream

```

<Dispatch: public>+≡
    public :: dispatch_eio

<Dispatch: procedures>+≡
    subroutine dispatch_eio (eio, method, global)
        class(eio_t), intent(inout), allocatable :: eio
        type(string_t), intent(in) :: method
        type(rt_data_t), intent(in) :: global
        logical :: keep_beams
        select case (char (method))
        case ("raw")
            allocate (eio_raw_t :: eio)
        case ("lhef")
            allocate (eio_lhef_t :: eio)
            select type (eio)
            type is (eio_lhef_t)
                keep_beams = &
                    var_list_get_lval (global%var_list, var_str ("?keep_beams"))
                call eio%set_parameters (keep_beams)
            end select
        case ("hepmc")
            allocate (eio_hepmc_t :: eio)
            select type (eio)
            type is (eio_lhef_t)
                keep_beams = &
                    var_list_get_lval (global%var_list, var_str ("?keep_beams"))
                call eio%set_parameters (keep_beams)

```

```

        end select
    case ("weight_stream")
        allocate (eio_weights_t :: eio)
    case default
        call msg_fatal ("Event I/O method '" // char (method) &
            // "' not implemented")
    end select
end subroutine dispatch_eio

```

### 19.6.9 QCD coupling

Allocate the `alpha` (running coupling) component of the `qcd` block with a concrete implementation, depending on the variable settings in the `global` record.

If a fixed  $\alpha_s$  is requested, we do not allocate the `qcd%alpha` object. In this case, the matrix element code will just take the model parameter as-is, which implies fixed  $\alpha_s$ . If the object is allocated, the  $\alpha_s$  value is computed and updated for each matrix-element call.

```

<Dispatch: public>+≡
    public :: dispatch_qcd

<Dispatch: procedures>+≡
    subroutine dispatch_qcd (qcd, global)
        type(qcd_t), intent(inout) :: qcd
        type(rt_data_t), intent(inout), target :: global
        type(var_list_t), pointer :: var_list
        logical :: fixed, from_mz, from_pdf_builtin
        real(default) :: mz, alpha_val, lambda
        integer :: nf, order
        type(string_t) :: pdfset
        var_list => global%var_list
        fixed = &
            var_list_get_lval (var_list, var_str ("?alpha_s_is_fixed"))
        from_mz = &
            var_list_get_lval (var_list, var_str ("?alpha_s_from_mz"))
        from_pdf_builtin = &
            var_list_get_lval (var_list, var_str ("?alpha_s_from_pdf_builtin"))
        pdfset = &
            var_list_get_sval (var_list, var_str ("pdf_builtin_set"))
        lambda = &
            var_list_get_rval (var_list, var_str ("lambda_qcd"))
        nf = &
            var_list_get_ival (var_list, var_str ("alpha_s_nf"))
        order = &
            var_list_get_ival (var_list, var_str ("alpha_s_order"))
        var_list => model_get_var_list_ptr (global%model)
        if (var_list_exists (var_list, var_str ("mZ"))) then
            mz = var_list_get_rval (var_list, var_str ("mZ"))
        else
            mz = MZ_REF
        end if
        if (var_list_exists (var_list, var_str ("alphas"))) then
            alpha_val = var_list_get_rval (var_list, var_str ("alphas"))

```

```

else
    alpha_val = ALPHA_QCD_MZ_REF
end if
if (allocated (qcd%alpha)) deallocate (qcd%alpha)
if (fixed) then
!    allocate (alpha_qcd_fixed_t :: qcd%alpha)
else if (from_mz) then
    allocate (alpha_qcd_from_scale_t :: qcd%alpha)
else if (from_pdf_builtin) then
    allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
else
    allocate (alpha_qcd_from_lambda_t :: qcd%alpha)
end if
if (allocated (qcd%alpha)) then
    select type (alpha => qcd%alpha)
    type is (alpha_qcd_fixed_t)
        alpha%val = alpha_val
    type is (alpha_qcd_from_scale_t)
        alpha%mu_ref = mz
        alpha%ref = alpha_val
        alpha%order = order
        alpha%nf = nf
    type is (alpha_qcd_from_lambda_t)
        alpha%lambda = lambda
        alpha%order = order
        alpha%nf = nf
    type is (alpha_qcd_pdf_builtin_t)
        call alpha%init (global%pdf_builtin_status, pdfset, &
            global%os_data%pdf_builtin_datapath)
    end select
!    else
!        call msg_bug ("dispatch_qcd: could not allocate QCD data")
    end if
end subroutine dispatch_qcd

```

### 19.6.10 Test

This is the master for calling self-test procedures.

```

<Dispatch: public>+≡
    public :: dispatch_test

<Dispatch: tests>≡
    subroutine dispatch_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Dispatch: execute tests>
end subroutine dispatch_test

```

### Select type: process definition

```

<Dispatch: execute tests>≡

```

```

call test (dispatch_1, "dispatch_1", &
          "process configuration method", &
          u, results)
(Dispatch: tests)+=≡
subroutine dispatch_1 (u)
  integer, intent(in) :: u
  type(string_t), dimension(2) :: prt_in, prt_out
  type(rt_data_t), target :: global
  class(prc_core_def_t), allocatable :: core_def

  write (u, "(A)")  "* Test output: dispatch_1"
  write (u, "(A)")  "* Purpose: select process configuration method"
  write (u, "(A)")

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
    .false., is_known = .true.)

  prt_in = [var_str ("a"), var_str ("b")]
  prt_out = [var_str ("c"), var_str ("d")]

  write (u, "(A)")  "* Allocate core_def as prc_test_def"

  call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known = .true.)
  call dispatch_core_def (core_def, prt_in, prt_out, global)
  select type (core_def)
  type is (prc_test_def_t)
    call core_def%write (u)
  end select

  deallocate (core_def)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate core_def as omega_def"
  write (u, "(A)")

  call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("omega"), is_known = .true.)
  call dispatch_core_def (core_def, prt_in, prt_out, global)
  select type (core_def)
  type is (omega_def_t)
    call core_def%write (u)
  end select

  call global%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_1"

end subroutine dispatch_1

```



## Select type: process core

```
<Dispatch: execute tests>+≡
  call test (dispatch_2, "dispatch_2", &
    "process core", &
    u, results)

<Dispatch: tests>+≡
  subroutine dispatch_2 (u)
    integer, intent(in) :: u
    type(string_t), dimension(2) :: prt_in, prt_out
    type(rt_data_t), target :: global
    class(prc_core_def_t), allocatable :: core_def
    class(prc_core_t), allocatable :: core
    type(model_t), pointer :: model

    write (u, "(A)")  "* Test output: dispatch_2"
    write (u, "(A)")  "*   Purpose: select process configuration method"
    write (u, "(A)")  "               and allocate process core"
    write (u, "(A)")

    call syntax_model_file_init ()
    call global%global_init ()

    prt_in = [var_str ("a"), var_str ("b")]
    prt_out = [var_str ("c"), var_str ("d")]

    write (u, "(A)")  "* Allocate core as test_t"
    write (u, "(A)")

    call var_list_set_string (global%var_list, var_str ("method"), &
      var_str ("unit_test"), is_known = .true.)
    call dispatch_core_def (core_def, prt_in, prt_out, global)
    call dispatch_core (core, core_def)
    select type (core)
    type is (test_t)
      call core%write (u)
    end select

    deallocate (core)
    deallocate (core_def)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate core as prc_omega_t"
    write (u, "(A)")

    call var_list_set_string (global%var_list, var_str ("method"), &
      var_str ("omega"), is_known = .true.)
    call dispatch_core_def (core_def, prt_in, prt_out, global)

    call model_list_read_model (var_str ("Test"), var_str ("Test.mdl"), &
      global%os_data, model)

    call var_list_set_log (global%var_list, &
      var_str ("?helicity_selection_active"), &
```

```

        .true., is_known = .true.)
call var_list_set_real (global%var_list, &
    var_str ("helicity_selection_threshold"), &
    1e9_default, is_known = .true.)
call var_list_set_int (global%var_list, &
    var_str ("helicity_selection_cutoff"), &
    10, is_known = .true.)

call dispatch_core (core, core_def, model, global%get_helicity_selection ())
call core_def%allocate_driver (core%driver, var_str (""))

select type (core)
type is (prc_omega_t)
    call core%write (u)
end select

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_2"

end subroutine dispatch_2

```

### Select type: integrator core

```

<Dispatch: execute tests>+≡
call test (dispatch_3, "dispatch_3", &
    "integration method", &
    u, results)

<Dispatch: tests>+≡
subroutine dispatch_3 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global
    class(mci_t), allocatable :: mci
    type(string_t) :: process_id

    write (u, "(A)")  "* Test output: dispatch_3"
    write (u, "(A)")  "* Purpose: select integration method"
    write (u, "(A)")

    call global%global_init ()
    process_id = "dispatch_3"

    write (u, "(A)")  "* Allocate MCI as midpoint_t"
    write (u, "(A)")

    call var_list_set_string (global%var_list, &
        var_str ("integration_method"), &
        var_str ("midpoint"), is_known = .true.)
    call dispatch_mci (mci, global, process_id)

```

```

select type (mci)
type is (mci_midpoint_t)
    call mci%write (u)
end select

call mci%final ()
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Allocate MCI as vamp_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
    var_str ("integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_set_int (global%var_list, var_str ("threshold_calls"), &
    1, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_calls_per_channel"), &
    2, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_calls_per_bin"), &
    3, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_bins"), &
    4, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("max_bins"), &
    5, is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?stratified"), &
    .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .false., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("channel_weights_power"), &
    4._default, is_known = .true.)

call dispatch_mci (mci, global, process_id)
select type (mci)
type is (mci_vamp_t)
    call mci%write (u)
end select

call mci%final ()
deallocate (mci)

call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_3"

end subroutine dispatch_3

```

## Select type: phase-space configuration object

```

<Dispatch: execute tests>+≡
call test (dispatch_4, "dispatch_4", &
    "phase-space configuration", &

```

```

        u, results)
<Dispatch: tests>+≡
subroutine dispatch_4 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global
    class(phs_config_t), allocatable :: phs

    write (u, "(A)")  "* Test output: dispatch_4"
    write (u, "(A)")  "*   Purpose: select phase-space configuration method"
    write (u, "(A)")

    call global%global_init ()

    write (u, "(A)")  "* Allocate PHS as phs_single_t"
    write (u, "(A)")

    call var_list_set_string (global%var_list, &
        var_str ("phs_method"), &
        var_str ("single"), is_known = .true.)
    call dispatch_phs (phs, global)
    call phs%write (u)

    call phs%final ()
    deallocate (phs)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate PHS as phs_wood_t"
    write (u, "(A)")

    call var_list_set_string (global%var_list, &
        var_str ("phs_method"), &
        var_str ("wood"), is_known = .true.)
    call dispatch_phs (phs, global)
    call phs%write (u)

    call phs%final ()

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_4"

end subroutine dispatch_4

```

### Select type: random number generator

```

<Dispatch: execute tests>+≡
    call test (dispatch_5, "dispatch_5", &
        "random-number generator", &
        u, results)
<Dispatch: tests>+≡
subroutine dispatch_5 (u)

```

```

integer, intent(in) :: u
type(rt_data_t), target :: global
class(rng_t), allocatable :: rng

write (u, "(A)")  "* Test output: dispatch_5"
write (u, "(A)")  "*   Purpose: select random-number generator"
write (u, "(A)")

call global%global_init ()

write (u, "(A)")  "* Allocate RNG as rng_test_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
    var_str ("rng_method"), &
    var_str ("unit_test"), is_known = .true.)
call dispatch_rng (rng, global)
select type (rng)
type is (rng_test_t)
    write (u, "(3x,A)")  "rng is rng_test_t"
end select

deallocate (rng)

write (u, "(A)")
write (u, "(A)")  "* Allocate RNG as rng_tao_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
    var_str ("rng_method"), &
    var_str ("tao"), is_known = .true.)
call dispatch_rng (rng, global)
select type (rng)
type is (rng_tao_t)
    write (u, "(3x,A)")  "rng is rng_tao_t"
end select

call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_5"

end subroutine dispatch_5

```

## Phase-space configuration with file

```

<Dispatch: execute tests>+≡
    call test (dispatch_6, "dispatch_6", &
        "configure phase space using file", &
        u, results)
<Dispatch: tests>+≡
    subroutine dispatch_6 (u)

```

```

integer, intent(in) :: u
type(rt_data_t), target :: global
type(model_t), pointer :: model
type(os_data_t) :: os_data
type(process_constants_t) :: process_data
class(phs_config_t), allocatable :: phs
integer :: u_phs

write (u, "(A)")  "* Test output: dispatch_6"
write (u, "(A)")  "* Purpose: select 'wood' phase-space &
                  &for a test process"
write (u, "(A)")  "*                  and read phs configuration from file"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call global%global_init ()

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list_read_model (var_str ("Test"), &
                           var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

call init_test_process_data (var_str ("dispatch_6"), process_data)

write (u, "(A)")  "* Write phase-space file"

u_phs = free_unit ()
open (u_phs, file = "dispatch_6.phs", action = "write", status = "replace")
call write_test_phs_file (u_phs, var_str ("dispatch_6"))
close (u_phs)

write (u, "(A)")
write (u, "(A)")  "* Allocate PHS as phs_wood_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
                          var_str ("phs_method"), &
                          var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, &
                          var_str ("phs_file"), &
                          var_str ("dispatch_6.phs"), is_known = .true.)
call dispatch_phs (phs, global)

call phs%init (process_data)
call phs%configure (sqrts = 1000._default)

call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_config_t)

```

```

        call phs%write_forest (u)
    end select

    call phs%final ()

    call global%final ()
    call model_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_6"

end subroutine dispatch_6

```

### Select type: structure-function data

```

<Dispatch: execute tests>+≡
    call test (dispatch_7, "dispatch_7", &
        "structure-function data", &
        u, results)

<Dispatch: tests>+≡
    subroutine dispatch_7 (u)
        integer, intent(in) :: u
        type(rt_data_t), target :: global
        type(os_data_t) :: os_data
        type(flavor_t) :: flv
        type(string_t) :: sf_method
        class(sf_data_t), allocatable :: data

        write (u, "(A)")  "* Test output: dispatch_7"
        write (u, "(A)")  "* Purpose: select and configure &
            &structure function data"
        write (u, "(A)")

        call global%global_init ()

        call os_data_init (os_data)
        call syntax_model_file_init ()
        call model_list_read_model (var_str ("QCD"), &
            var_str ("QCD.mdl"), os_data, global%model)

        call flavor_init (flv, PROTON, global%model)

        call reset_interaction_counter ()
        call beam_data_init_sqrts (global%beam_data, &
            14000._default, [flv, flv])

        write (u, "(1x,A,L1)")  "Initialized = ", &
            pdf_builtin_status_is_initialized (global%pdf_builtin_status)
        write (u, "(A)")

        write (u, "(A)")  "* Allocate data as sf_pdf_builtin_t"
        write (u, "(A)")
    end subroutine dispatch_7

```

```

sf_method = "pdf_builtin"
call dispatch_sf_data (data, sf_method, [1], global)
call data%write (u)

write (u, "(A)")
write (u, "(1x,A,L1)") "Initialized = ", &
    pdf_builtin_status_is_initialized (global%pdf_builtin_status)
write (u, "(A)")
write (u, "(1x,A,99(1x,I0))") "PDG(out) = ", data%get_pdg_out ()

deallocate (data)

write (u, "(A)")
write (u, "(A)")  "* Reset and allocate data for different PDF set"
write (u, "(A)")

call pdf_builtin_status_reset (global%pdf_builtin_status)

write (u, "(1x,A,L1)") "Initialized = ", &
    pdf_builtin_status_is_initialized (global%pdf_builtin_status)
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("pdf_builtin_set"), &
    var_str ("CTEQ6M"), is_known = .true.)
sf_method = "pdf_builtin"
call dispatch_sf_data (data, sf_method, [1], global)
call data%write (u)

write (u, "(A)")
write (u, "(1x,A,L1)") "Initialized = ", &
    pdf_builtin_status_is_initialized (global%pdf_builtin_status)
write (u, "(A)")
write (u, "(1x,A,99(1x,I0))") "PDG(out) = ", data%get_pdg_out ()

deallocate (data)

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_7"

end subroutine dispatch_7

```

## Beam structure

```

<Dispatch: execute tests>+≡
    call test (dispatch_8, "dispatch_8", &
        "beam structure", &
        u, results)
<Dispatch: tests>+≡

```



```

subroutine dispatch_8 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global
  type(os_data_t) :: os_data
  type(flavor_t) :: flv
  type(sf_config_t), dimension(:), allocatable :: sf_config
  type(sf_channel_t), dimension(:), allocatable :: sf_channel
  integer :: i

  write (u, "(A)")  "* Test output: dispatch_8"
  write (u, "(A)")  "* Purpose: configure a structure-function chain"
  write (u, "(A)")

  call global%global_init ()

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list_read_model (var_str ("QCD"), &
    var_str ("QCD.mdl"), os_data, global%model)

  write (u, "(A)")  "* Allocate LHC beams with PDF builtin"
  write (u, "(A)")

  call flavor_init (flv, PROTON, global%model)

  call reset_interaction_counter ()
  call beam_data_init_sqrts (global%beam_data, &
    14000._default, [flv, flv])

  call global%beam_structure%init ([1], global%var_list)
  call global%beam_structure%set_entry (1, 1, var_str ("pdf_builtin"))

  call dispatch_sf_config (sf_config, sf_channel, global)

  do i = 1, size (sf_config)
    call sf_config(i)%write (u)
  end do
  write (u, "(1x,A)")  "Mapping configuration:"
  do i = 1, size (sf_channel)
    write (u, "(2x)", advance = "no")
    call sf_channel(i)%write (u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call global%final ()
  call model_list_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_8"

end subroutine dispatch_8

```

## Event I/O

```
<Dispatch: execute tests>+≡
  call test (dispatch_9, "dispatch_9", &
    "event I/O", &
    u, results)

<Dispatch: tests>+≡
  subroutine dispatch_9 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global
    class(eio_t), allocatable :: eio

    write (u, "(A)")  "* Test output: dispatch_9"
    write (u, "(A)")  "* Purpose: allocate an event I/O (eio) stream"
    write (u, "(A)")

    call global%global_init ()

    write (u, "(A)")  "* Allocate as raw"
    write (u, "(A)")

    call dispatch_eio (eio, var_str ("raw"), global)

    call eio%write (u)

    call eio%final ()
    deallocate (eio)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate as LHEF:"
    write (u, "(A)")

    call dispatch_eio (eio, var_str ("lhef"), global)

    call eio%write (u)

    call eio%final ()
    deallocate (eio)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate as HepMC:"
    write (u, "(A)")

    call dispatch_eio (eio, var_str ("hepmc"), global)

    call eio%write (u)

    call eio%final ()
    deallocate (eio)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate as weight_stream"
    write (u, "(A)")
```

```

call dispatch_eio (eio, var_str ("weight_stream"), global)

call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_9"

end subroutine dispatch_9

```

## Update process core parameters

This test dispatches a process core, temporarily modifies parameters, then restores the original.

```

<Dispatch: execute tests>+≡
call test (dispatch_10, "dispatch_10", &
          "process core update", &
          u, results)

<Dispatch: tests>+≡
subroutine dispatch_10 (u)
integer, intent(in) :: u
type(string_t), dimension(2) :: prt_in, prt_out
type(rt_data_t), target :: global
class(prc_core_def_t), allocatable :: core_def
class(prc_core_t), allocatable :: core, saved_core
type(model_t), pointer :: model
type(var_list_t), pointer :: var_list

write (u, "(A)")  "* Test output: dispatch_10"
write (u, "(A)")  "* Purpose: select process configuration method,"
write (u, "(A)")  "          allocate process core,"
write (u, "(A)")  "          temporarily reset parameters"
write (u, "(A)")

call syntax_model_file_init ()
call global%global_init ()

prt_in = [var_str ("a"), var_str ("b")]
prt_out = [var_str ("c"), var_str ("d")]

write (u, "(A)")  "* Allocate core as prc_omega_t"
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("method"), &
                          var_str ("omega"), is_known = .true.)
call dispatch_core_def (core_def, prt_in, prt_out, global)

```

```

call model_list_read_model (var_str ("Test"), var_str ("Test.mdl"), &
    global%os_data, model)

call dispatch_core (core, core_def, model)
call core_def%allocate_driver (core%driver, var_str (""))

select type (core)
type is (prc_omega_t)
    call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Update core with modified model and helicity selection"
write (u, "(A)")

var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, var_str ("gy"), 2._default, &
    is_known = .true.)
call model_parameters_update (model)

call var_list_set_log (global%var_list, &
    var_str ("?helicity_selection_active"), &
    .true., is_known = .true.)
call var_list_set_real (global%var_list, &
    var_str ("helicity_selection_threshold"), &
    2e10_default, is_known = .true.)
call var_list_set_int (global%var_list, &
    var_str ("helicity_selection_cutoff"), &
    5, is_known = .true.)

call dispatch_core_update (core, model, global%get_helicity_selection (), &
    saved_core = saved_core)
select type (core)
type is (prc_omega_t)
    call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Restore core from save"
write (u, "(A)")

call dispatch_core_restore (core, saved_core)
select type (core)
type is (prc_omega_t)
    call core%write (u)
end select

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_10"

end subroutine dispatch_10

```

## QCD Coupling

This test dispatches an qcd object, which is used to compute the (running) coupling by one of several possible methods.

```
<Dispatch: execute tests>+≡
  call test (dispatch_11, "dispatch_11", &
    "QCD coupling", &
    u, results)

<Dispatch: tests>+≡
  subroutine dispatch_11 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global
    type(qcd_t) :: qcd
    type(var_list_t), pointer :: model_vars

    write (u, "(A)")  "* Test output: dispatch_11"
    write (u, "(A)")  "* Purpose: select QCD coupling formula"
    write (u, "(A)")

    call syntax_model_file_init ()
    call global%global_init ()
    call model_list_read_model (var_str ("SM"), var_str ("SM.mdl"), &
      global%os_data, global%model)
    model_vars => model_get_var_list_ptr (global%model)

    write (u, "(A)")  "* Don't allocate: implies fixed alpha_s"
    write (u, "(A)")

    call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
      .true., is_known = .true.)
    call dispatch_qcd (qcd, global)
    call qcd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate alpha_s as running (built-in)"
    write (u, "(A)")

    call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
      .false., is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?alpha_s_from_mz"), &
      .true., is_known = .true.)
    call var_list_set_int &
      (global%var_list, var_str ("alpha_s_order"), 1, is_known = .true.)
    call var_list_set_real &
      (model_vars, var_str ("alphas"), 0.1234_default, &
        is_known=.true.)
    call var_list_set_real &
      (model_vars, var_str ("mZ"), 91.234_default, &
        is_known=.true.)
    call dispatch_qcd (qcd, global)
    call qcd%write (u)
```

```

write (u, "(A)")
write (u, "(A)")  "* Allocate alpha_s as running (built-in, Lambda defined)"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
    .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?alpha_s_from_mz"), &
    .false., is_known = .true.)
call var_list_set_real &
    (global%var_list, var_str ("lambda_qcd"), 250.e-3_default, &
    is_known=.true.)
call var_list_set_int &
    (global%var_list, var_str ("alpha_s_order"), 2, is_known = .true.)
call var_list_set_int &
    (global%var_list, var_str ("alpha_s_nf"), 4, is_known = .true.)
call dispatch_qcd (qcd, global)
call qcd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Allocate alpha_s as running (using builtin PDF set)"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
    .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?alpha_s_from_mz"), &
    .false., is_known = .true.)
call var_list_set_log &
    (global%var_list, var_str ("?alpha_s_from_pdf_builtin"), &
    .true., is_known = .true.)
call dispatch_qcd (qcd, global)
call qcd%write (u)

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_11"

end subroutine dispatch_11

```

## 19.7 Process Configuration

This module communicates between the toplevel command structure with its runtime data set and the process-library handling modules which collect the definition of individual processes. Its primary purpose is to select from the available matrix-element generating methods and configure the entry in the process library accordingly.

`<process_configurations.f90>≡`  
*<File header>*

```

module process_configurations

  <Use strings>
    use diagnostics !NODEP!
    use unit_tests
    use variables
    use models
    use prc_core_def
    use particle_specifiers
    use process_libraries
    use prclib_stacks
    use prc_test
    use prc_omega
    use rt_data
    use dispatch

  <Standard module head>

  <Process configurations: public>

  <Process configurations: types>

  contains

  <Process configurations: procedures>

  <Process configurations: tests>

end module process_configurations

```

### 19.7.1 Data Type

```

<Process configurations: public>≡
  public :: process_configuration_t

<Process configurations: types>≡
  type :: process_configuration_t
    type(process_def_entry_t), pointer :: entry => null ()
    type(string_t) :: id
  contains
    <Process configurations: process configuration: TBP>
  end type process_configuration_t

```

Initialize a process. We only need the name, the number of incoming particles, and the number of components.

```

<Process configurations: process configuration: TBP>≡
  procedure :: init => process_configuration_init

<Process configurations: procedures>≡
  subroutine process_configuration_init &
    (config, prc_name, n_in, n_components, global)
    class(process_configuration_t), intent(out) :: config
    type(string_t), intent(in) :: prc_name
    integer, intent(in) :: n_in

```

```

integer, intent(in) :: n_components
type(rt_data_t), intent(in) :: global
type(model_t), pointer :: model
model => global%model
config%id = prc_name
allocate (config%entry)
call config%entry%init (prc_name, &
    model = model, n_in = n_in, n_components = n_components)
end subroutine process_configuration_init

```

Initialize a process component. The details depend on the process method, which determines the type of the process component core. We set the incoming and outgoing particles (as strings, to be interpreted by the process driver). All other information is taken from the variable list.

The dispatcher gets only the names of the particles. The process component definition gets the complete specifiers which contains a polarization flag and names of decay processes, where applicable.

```

(Process configurations: process configuration: TBP)+≡
    procedure :: setup_component => process_configuration_setup_component

(Process configurations: procedures)+≡
    subroutine process_configuration_setup_component &
        (config, i_component, prt_in, prt_out, global)
        class(process_configuration_t), intent(inout) :: config
        integer, intent(in) :: i_component
        type(prt_spec_t), dimension(:), intent(in) :: prt_in
        type(prt_spec_t), dimension(:), intent(in) :: prt_out
        type(rt_data_t), intent(in) :: global
        type(string_t), dimension(:), allocatable :: prt_str_in
        type(string_t), dimension(:), allocatable :: prt_str_out
        class(prc_core_def_t), allocatable :: core_def
        type(string_t) :: method
        integer :: i
        allocate (prt_str_in (size (prt_in)))
        allocate (prt_str_out (size (prt_out)))
        forall (i = 1:size (prt_in)) prt_str_in(i) = prt_in(i)% get_name ()
        forall (i = 1:size (prt_out)) prt_str_out(i) = prt_out(i)%get_name ()
        call dispatch_core_def (core_def, prt_str_in, prt_str_out, global)
        method = var_list_get_sval (global%var_list, var_str ("method"))
        call config%entry%import_component (i_component, &
            n_out = size (prt_out), &
            prt_in = prt_in, &
            prt_out = prt_out, &
            method = method, &
            variant = core_def)
    end subroutine process_configuration_setup_component

```

Record a process configuration: append it to the currently selected process definition library.

```

(Process configurations: process configuration: TBP)+≡
    procedure :: record => process_configuration_record

```



```

<Process configurations: procedures>+≡
  subroutine process_configuration_record (config, global)
    class(process_configuration_t), intent(inout) :: config
    type(rt_data_t), intent(inout) :: global
    if (associated (global%prclib)) then
      call global%prclib%open ()
      call global%prclib%append (config%entry)
      call msg_message ("Process library '" &
        // char (global%prclib%get_name ()) &
        // "' : recorded process '" // char (config%id) // "'")
    else
      call msg_fatal ("Recording process '" // char (config%id) &
        // "' : active process library undefined")
    end if
  end subroutine process_configuration_record

```

## 19.7.2 Test

This is the master for calling self-test procedures.

```

<Process configurations: public>+≡
  public :: process_configurations_test

<Process configurations: tests>≡
  subroutine process_configurations_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <Process configurations: execute tests>
  end subroutine process_configurations_test

```

## Minimal setup

The workflow for setting up a minimal process configuration with the test matrix element method.

We wrap this in a public procedure, so we can reuse it in later modules. The procedure prepares a process definition list for two processes (one **prc\_test** and one **omega** type) and appends this to the process library stack in the global data set.

The **mode** argument determines which processes to build.

The **procname** argument replaces the predefined **procname(s)**.

```

<Process configurations: public>+≡
  public :: prepare_test_library

<Process configurations: tests>+≡
  subroutine prepare_test_library (global, libname, mode, procname)
    type(rt_data_t), intent(inout), target :: global
    type(string_t), intent(in) :: libname
    integer, intent(in) :: mode
    type(string_t), intent(in), dimension(:), optional :: procname

    type(var_list_t), pointer :: model_vars
    type(prclib_entry_t), pointer :: lib

```

```

type(string_t) :: prc_name
type(string_t), dimension(:), allocatable :: prt_in, prt_out
integer :: n_components
type(process_configuration_t) :: prc_config

allocate (lib)
call lib%init (libname)
call global%add_prclib (lib)

if (btest (mode, 0)) then
    call model_list_read_model (var_str ("Test"), &
        var_str ("Test.mdl"), global%os_data, global%model)
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_init_copies &
        (global%var_list, model_vars)
end if

if (btest (mode, 1)) then
    call model_list_read_model (var_str ("QED"), &
        var_str ("QED.mdl"), global%os_data, global%model)
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_init_copies &
        (global%var_list, model_vars)
end if

if (btest (mode, 0)) then

    global%model => model_list_get_model_ptr (var_str ("Test"))
    call var_list_set_string (global%var_list, var_str ("model_name"), &
        model_get_name (global%model), is_known=.true.)
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_synchronize &
        (global%var_list, model_vars, reset_pointers = .true.)

    if (present (procname)) then
        prc_name = procname(1)
    else
        prc_name = "prc_config_a"
    end if
    n_components = 1
    allocate (prt_in (2), prt_out (2))
    prt_in = [var_str ("s"), var_str ("s")]
    prt_out = [var_str ("s"), var_str ("s")]

    call var_list_set_string (global%var_list, var_str ("method"),&
        var_str ("unit_test"), is_known = .true.)

    call prc_config%init (prc_name, size (prt_in), n_components, global)
    call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global)
    call prc_config%record (global)

    deallocate (prt_in, prt_out)

```

```

end if

if (btest (mode, 1)) then

    global%model => model_list_get_model_ptr (var_str ("QED"))
    call var_list_set_string (global%var_list, var_str ("model_name"), &
        model_get_name (global%model), is_known=.true.)
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_synchronize &
        (global%var_list, model_vars, reset_pointers = .true.)

    if (present (procname)) then
        prc_name = procname(2)
    else
        prc_name = "prc_config_b"
    end if
    n_components = 1
    allocate (prt_in (2), prt_out (2))
    prt_in = [var_str ("e+"), var_str ("e-")]
    prt_out = [var_str ("m+"), var_str ("m-")]

    call var_list_set_string (global%var_list, var_str ("method"), &
        var_str ("omega"), is_known = .true.)

    call prc_config%init (prc_name, size (prt_in), n_components, global)
    call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global)
    call prc_config%record (global)

    deallocate (prt_in, prt_out)

end if

end subroutine prepare_test_library

```

The actual test: the previous procedure with some prelude and postlude. In the global variable list, just before printing we reset the variables where the value may depend on the system and run environment.

```

<Process configurations: execute tests>≡
    call test (process_configurations_1, "process_configurations_1", &
        "test processes", &
        u, results)

<Process configurations: tests>+≡
    subroutine process_configurations_1 (u)
        integer, intent(in) :: u
        type(rt_data_t), target :: global

        write (u, "(A)")  "* Test output: process_configurations_1"
        write (u, "(A)")  "* Purpose: configure test processes"
        write (u, "(A)")

        call syntax_model_file_init ()

```

```

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)

write (u, "(A)")  "* Configure processes as prc_test, model Test"
write (u, "(A)")  "*                               and omega, model QED"
write (u, *)

call prepare_test_library (global, var_str ("prc_config_lib_1"), 3)

global%os_data%fc = "Fortran-compiler"
global%os_data%fcflags = "Fortran-flags"

call global%write_libraries (u)

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_configurations_1"

end subroutine process_configurations_1

```

## O'MEGA options

Slightly extended example where we pass O'MEGA options to the library. The `prepare_test_library` contents are spelled out.

```

<Process configurations: execute tests>+≡
    call test (process_configurations_2, "process_configurations_2", &
        "omega options", &
        u, results)

<Process configurations: tests>+≡
subroutine process_configurations_2 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    type(string_t) :: libname
    type(var_list_t), pointer :: model_vars
    type(prclib_entry_t), pointer :: lib
    type(string_t) :: prc_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    integer :: n_components
    type(process_configuration_t) :: prc_config

    write (u, "(A)")  "* Test output: process_configurations_2"
    write (u, "(A)")  "* Purpose: configure test processes with options"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

```

```

write (u, "(A)")  "* Configure processes as omega, model QED"
write (u, *)

libname = "prc_config_lib_2"

allocate (lib)
call lib%init (libname)
call global%add_prclib (lib)

call model_list_read_model (var_str ("QED"), &
    var_str ("QED.mdl"), global%os_data, global%model)
model_vars => model_get_var_list_ptr (global%model)
call var_list_init_copies &
    (global%var_list, model_vars)

global%model => model_list_get_model_ptr (var_str ("QED"))
call var_list_set_string (global%var_list, var_str ("model_name"), &
    model_get_name (global%model), is_known=.true.)
model_vars => model_get_var_list_ptr (global%model)
call var_list_synchronize &
    (global%var_list, model_vars, reset_pointers = .true.)

prc_name = "prc_config_c"
n_components = 2
allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e+"), var_str ("e-")]
prt_out = [var_str ("m+"), var_str ("m-")]

call var_list_set_string (global%var_list, var_str ("method"), &
    var_str ("omega"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
    .false., is_known = .true.)

call prc_config%init (prc_name, size (prt_in), n_components, global)

call var_list_set_log (global%var_list, var_str ("?report_progress"), &
    .true., is_known = .true.)
call prc_config%setup_component (1, &
    new_prt_spec (prt_in), new_prt_spec (prt_out), global)

call var_list_set_log (global%var_list, var_str ("?report_progress"), &
    .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
    .true., is_known = .true.)
call var_list_set_string (global%var_list, var_str ("restrictions"), &
    var_str ("3+4~A"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("omega_flags"), &
    var_str ("-fusion:progress_file omega_prc_config.log"), &
    is_known = .true.)
call prc_config%setup_component (2, &
    new_prt_spec (prt_in), new_prt_spec (prt_out), global)

call prc_config%record (global)

```

```

deallocate (prt_in, prt_out)

global%os_data%fc = "Fortran-compiler"
global%os_data%fcflags = "Fortran-flags"

call global%write_vars (u, [ &
    var_str ("model_name"), &
    var_str ("method"), &
    var_str ("report_progress"), &
    var_str ("restrictions"), &
    var_str ("omega_flags")])
write (u, "(A)")
call global%write_libraries (u)

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_configurations_2"

end subroutine process_configurations_2

```

## 19.8 Compilation

This module manages compilation and loading of process libraries. It is needed as a separate module because integration depends on it.

```

<compilations.f90>≡
  <File header>

  module compilations

    <Use strings>
    use diagnostics !NODEP!
    use os_interface
    use unit_tests
    use variables
    use models
    use process_libraries
    use prclib_stacks
    use rt_data
    use process_configurations

    <Standard module head>

    <Compilations: public>

    <Compilations: types>

    <Compilations: interfaces>

    contains

```

*<Compilations: procedures>*

*<Compilations: tests>*

end module compilations

### 19.8.1 The data type

This data type serves as a container that collects all necessary data; the procedures `compile` and `load` operate on it internally.

*<Compilations: types>*≡

```
type :: compilation_t
  type(string_t) :: libname
  type(process_library_t), pointer :: lib => null ()
  logical :: recompile_library = .false.
  logical :: make_executable = .false.
  contains
    <Compilations: compilation: TBP>
end type compilation_t
```

Initialize:

*<Compilations: compilation: TBP>*≡

```
procedure :: init => compilation_init
```

*<Compilations: procedures>*≡

```
subroutine compilation_init (comp, libname, stack, var_list)
  class(compilation_t), intent(out) :: comp
  type(string_t), intent(in) :: libname
  type(prclib_stack_t), intent(inout) :: stack
  type(var_list_t), intent(in) :: var_list
  comp%libname = libname
  comp%lib => stack%get_library_ptr (comp%libname)
  if (.not. associated (comp%lib)) then
    call msg_fatal ("Process library '" // char (comp%libname) &
      // "' has not been declared.")
  end if
  comp%recompile_library = &
    var_list_get_lval (var_list, var_str ("?recompile_library"))
end subroutine compilation_init
```

Compile and load the current library. The `force` flag has the effect that we first delete any previous files, as far as accessible by the current makefile. It also guarantees that previous files not accessible by a makefile will be overwritten.

*<Compilations: compilation: TBP>*+≡

```
procedure :: compile_and_load => compilation_compile_and_load
```

*<Compilations: procedures>*+≡

```
subroutine compilation_compile_and_load (comp, os_data, force)
  class(compilation_t), intent(inout) :: comp
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: force
  if (associated (comp%lib)) then
```

```

        call msg_message ("Process library '" &
            // char (comp%libname) // "' : compiling ...")
        call comp%lib%configure ()
        call comp%lib%compute_md5sum ()
        call comp%lib%write_makefile (os_data, force)
        if (force) call comp%lib%clean (os_data, distclean = .false.)
        call comp%lib%write_driver (force)
        call comp%lib%update_status (os_data)
        call comp%lib%load (os_data)
        call msg_message ("Process library '" // char (comp%libname) &
            // "' : ... success.")
    end if
end subroutine compilation_compile_and_load

```

Make a stand-alone executable

*(XXX Compilations: procedures)≡*

```

subroutine compilations_make_executable (comp, os_data, exec_name, var_list)
    type(compilation_t), dimension(:), intent(in) :: comp
    type(os_data_t), intent(in) :: os_data
    type(string_t), intent(in) :: exec_name
    type(var_list_t), intent(in) :: var_list
    type(string_t) :: flags
    integer :: lib
    type(user_procs_t) :: user_procs
    call splice &
        (var_list_get_sval (var_list, var_str ("user_procs_cut")), &
            user_procs%cut)
    call splice &
        (var_list_get_sval (var_list, var_str ("user_procs_event_shape")), &
            user_procs%event_shape)
    call splice &
        (var_list_get_sval (var_list, var_str ("user_procs_obs1")), &
            user_procs%obs_real_unary)
    call splice &
        (var_list_get_sval (var_list, var_str ("user_procs_obs2")), &
            user_procs%obs_real_binary)
    call splice &
        (var_list_get_sval (var_list, var_str ("user_procs_sf")), &
            user_procs%sf)
    call write_library_manager (comp%libname, user_procs)
    call compile_library_manager (os_data)
    flags = ""
    do lib = 1, size (comp)
        flags = flags // get_modellibs_flags (comp(lib)%prc_lib, os_data)
    end do
    call link_executable (comp%libname, exec_name, flags, os_data)
contains
    subroutine splice (string, array)
        type(string_t), intent(in) :: string
        type(string_t), dimension(:), intent(out), allocatable :: array
        type(string_t) :: buffer, word, separator
        integer :: n_commas, i
        if (string /= "") then

```



```

        buffer = string
        n_commas = 0
        COUNT_COMMAS: do
            call split (buffer, word, ",", separator)
            if (len (separator) == 0) exit COUNT_COMMAS
            n_commas = n_commas + 1
        end do COUNT_COMMAS
        allocate (array (n_commas + 1))
        buffer = string
        ASSIGN_STRINGS: do i = 1, size (array)
            call split (buffer, word, ",")
            array(i) = adjustl (trim (word))
        end do ASSIGN_STRINGS
    else
        allocate (array (0))
    end if
end subroutine splice
end subroutine compilations_make_executable

```

## 19.8.2 API for compilation and loading

The `global` data set may actually be local to the caller. The compilation affects the library specified by its name if it is on the stack, but it does not reset the currently selected library.

```

<Compilations: public>≡
    public :: compile_library

<Compilations: procedures>+≡
    subroutine compile_library (libname, global)
        type(string_t), intent(in) :: libname
        type(rt_data_t), intent(inout), target :: global
        type(compilation_t) :: comp
        logical :: force
        force = var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
        call comp%init (libname, global%prclib_stack, global%var_list)
        call comp%compile_and_load (global%os_data, force)
    end subroutine compile_library

```

Compile the processes. Do not load the library (libraries), but link them to a separate executable.

```

<XXX Compilations: public>≡
    public :: compile_executable

<XXX Compilations: interfaces>≡
    interface compile_executable
        module procedure compile_executable0
        module procedure compile_executable1
    end interface

<XXX Compilations: procedures>+≡
    subroutine compile_executable0 (libname, exec_name, global)
        type(string_t), intent(in) :: libname, exec_name

```

```

    type(rt_data_t), intent(in) :: global
    type(compilation_t), dimension(1) :: comp
    call compilation_basic_init (comp(1), libname, global%var_list)
    call compilation_compile_and_link (comp(1), global%os_data)
    call compilations_make_executable &
        (comp, global%os_data, exec_name, global%var_list)
end subroutine compile_executable0

subroutine compile_executable1 (libname, exec_name, global)
    type(string_t), dimension(:), intent(in) :: libname
    type(string_t), intent(in) :: exec_name
    type(rt_data_t), intent(in) :: global
    type(compilation_t), dimension(size(libname)) :: comp
    integer :: lib
    do lib = 1, size (libname)
        call compilation_basic_init (comp(lib), libname(lib), global%var_list)
        call compilation_compile_and_link (comp(lib), global%os_data)
    end do
    call compilations_make_executable &
        (comp, global%os_data, exec_name, global%var_list)
end subroutine compile_executable1

```

Just load the library (libraries), no compile.

*<XXX Compilations: public>+≡*

```
public :: load_library
```

*<XXX Compilations: interfaces>+≡*

```
interface load_library
    module procedure load_library0
    module procedure load_library1
end interface

```

*<XXX Compilations: procedures>+≡*

```

subroutine load_library0 (libname, global, global_var_list, global_prc_lib)
    type(string_t), intent(in) :: libname
    type(rt_data_t), intent(in) :: global
    type(var_list_t), intent(inout) :: global_var_list
    type(process_library_t), pointer :: global_prc_lib
    type(compilation_t) :: comp
    call compilation_basic_init (comp, libname, global%var_list)
    call compilation_load_library (comp, global%os_data, global_var_list)
    global_prc_lib => comp%prc_lib
end subroutine load_library0

```

```

subroutine load_library1 (libname, global, global_var_list, global_prc_lib)
    type(string_t), dimension(:), intent(in) :: libname
    type(rt_data_t), intent(in) :: global
    type(var_list_t), intent(inout) :: global_var_list
    type(process_library_t), pointer :: global_prc_lib
    integer :: lib
    do lib = 1, size (libname)
        call load_library0 &
            (libname(lib), global, global_var_list, global_prc_lib)
    end do

```

```
end subroutine load_library1
```

### 19.8.3 Test

This is the master for calling self-test procedures.

```
<Compilations: public>+≡
  public :: compilations_test

<Compilations: tests>≡
  subroutine compilations_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    <Compilations: execute tests>
  end subroutine compilations_test
```

#### Intrinsic Matrix Element

Compile an intrinsic test matrix element (`prc_test` type).

Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

```
<Compilations: execute tests>≡
  call test (compilations_1, "compilations_1", &
    "intrinsic test processes", &
    u, results)

<Compilations: tests>+≡
  subroutine compilations_1 (u)
    integer, intent(in) :: u
    type(string_t) :: libname
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: compilations_1"
    write (u, "(A)")  "* Purpose: configure and compile test process"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    libname = "compilation_1"
    call prepare_test_library (global, libname, 1)

    call compile_library (libname, global)

    call global%write_libraries (u)

    call global%final ()
    call model_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_1"
```

```
end subroutine compilations_1
```

## External Matrix Element

Compile an external test matrix element (omega type)

```
<Compilations: execute tests>+≡
  call test (compilations_2, "compilations_2", &
    "external process (omega)", &
    u, results)

<Compilations: tests>+≡
  subroutine compilations_2 (u)
    integer, intent(in) :: u
    type(string_t) :: libname
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: compilations_2"
    write (u, "(A)")  "* Purpose: configure and compile test process"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
      .false., is_known = .true.)

    libname = "compilation_2"
    call prepare_test_library (global, libname, 2)

    call compile_library (libname, global)

    call global%write_libraries (u)

    call global%final ()
    call model_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_2"

  end subroutine compilations_2
```

## 19.9 Integration

This module manages phase space setup, matrix-element evaluation and integration, as far as it is not done by lower-level routines, in particular in the `processes` module.

```
<integrations.f90>≡
  <File header>
```

```

module integrations

  <Use kinds>
  <Use strings>
  <Use file utils>
  ! use limits, only: ITERATIONS_DEFAULT_LIST_SIZE !NODEP!
  use diagnostics !NODEP!
  use unit_tests
  ! use tao_random_numbers !NODEP!
  ! use pdf_builtin !NODEP!
  use os_interface
  use sm_qcd
  use ifiles
  use lexers
  use parser
  use flavors
  use variables
  use expressions
  use models
  use interactions
  use beams
  use sf_mappings
  use sf_base
  use phs_base
  use phs_forests
  use phs_wood
  use rng_base
  use mci_base
  use process_libraries
  use prc_core
  use processes
  use process_stacks
  ! use strfun_config
  use rt_data
  use dispatch
  ! use iterations
  use process_configurations
  use compilations

  <Standard module head>

  <Integrations: public>

  <Integrations: types>

  <Integrations: interfaces>

contains

  <Integrations: procedures>

  <Integrations: tests>

end module integrations

```

### 19.9.1 The integration type

This type holds all relevant data, the integration methods operates on this. In contrast to the `simulation_t` introduced later, the `integration_t` applies to a single process. We need not export it.

```
(Integrations: types)≡
  type :: integration_t
    private
      type(string_t) :: process_id
      type(string_t) :: run_id
      type(process_t), pointer :: process => null ()
      logical :: rebuild_phs = .false.
!     logical :: check_phs_file = .true.
!     type(string_t) :: phs_filename
!     type(string_t) :: phs_filename_out
!     type(string_t) :: phs_filename_vis
      logical :: phs_only = .false.
!     logical :: vis_channels = .false.
!     type(phs_parameters_t) :: phs_par
!     type(mapping_defaults_t) :: mapping_defaults
!     logical :: rebuild_grids = .false.
!     logical :: check_grid_file = .true.
!     type(grid_parameters_t) :: grid_parameters
!     type(string_t) :: grids_filename
!     logical :: use_best_grid = .true.
!     real(default) :: accuracy_goal = 0
!     real(default) :: abs_error_goal = 0
!     real(default) :: rel_error_goal = 0
!     logical :: helicity_selection_active = .false.
!     real(default) :: helicity_selection_threshold = -1
!     integer :: helicity_selection_cutoff = 1000
!     logical :: sqrts_known = .false.
!     real(default) :: sqrts = -1
!     real(default) :: alpha_s = -1
!     real(default) :: alpha = 0
!     integer :: n_events_for_me_test = 0
!     logical :: time_estimate = .false.
!     logical :: vis_history = .true.
!     type(string_t) :: history_filename
!     type(beam_data_t) :: beam_data
!     logical :: use_beams = .false.
!     type(sf_list_t), pointer :: sf_list => null ()
!     logical :: use_strfun = .false.
!     type(iterations_list_t) :: it_list
!     integer :: pass_on_file = 0
!     integer :: it_on_file = 0
!     type(md5sum_grids_t) :: md5sum
!     integer :: pass = 0
!     integer :: it = 0
!     type(string_t) :: log_filename
    contains
      (Integrations: integration: TBP)
end type integration_t
```

## 19.9.2 Initialization

Initialization, first part: Create and initialize a process entry and push it on the stack. Only the basic process data (ID, run ID, etc.) are configured, we postpone the rest.

```

<Integrations: integration: TBP>≡
    procedure :: create_process => integration_create_process

<Integrations: procedures>≡
    subroutine integration_create_process (intg, process_id, global) !, verbose)
        class(integration_t), intent(out) :: intg
        type(rt_data_t), target :: global
        type(string_t), intent(in) :: process_id
    !    logical, intent(in), optional :: verbose
        type(var_list_t), pointer :: var_list
    !    logical :: verb
        type(process_entry_t), pointer :: process_entry

        var_list => global%var_list
        intg%process_id = process_id
        intg%run_id = var_list_get_sval (var_list, var_str ("$_run_id"))

        allocate (process_entry)
        call process_entry%init (intg%process_id, intg%run_id, global%prclib, &
            global%os_data)
        call process_entry%set_var_list (global%var_list)
        call global%process_stack%push (process_entry)

    end subroutine integration_create_process

```

Initialize and check beam parameters; if we are using beams, beam masses must be consistent with the current model.

```

<XXX Integrations: procedures>≡
    subroutine integration_check_beam_data (intg, beam_data)
        type(integration_t), intent(inout) :: intg
        type(beam_data_t), intent(in) :: beam_data
        intg%beam_data = beam_data
        intg%use_beams = beam_data_are_valid (intg%beam_data)
        if (intg%use_beams) then
            if (.not. beam_data_masses_are_consistent (intg%beam_data)) then
                call msg_warning &
                    ("Masses of beam particle(s) differ from beam masses")
            end if
        end if
    end subroutine integration_check_beam_data

```

Initialize the process library: if the process library is not yet compiled, insert a compilation.

```

<XXX Integrations: procedures>+≡
    subroutine maybe_compile_library (global)
        type(rt_data_t), target :: global
        call process_library_update_status (global%prc_lib)
        if (.not. process_library_is_compiled (global%prc_lib)) then

```

```

        call compile_library (process_library_get_name (global%prc_lib), &
                             global, global%var_list, global%prc_lib)
    end if
end subroutine maybe_compile_library

```

Select the appropriate process object from the stack and complete its configuration.

```

<Integrations: integration: TBP>+≡
    procedure :: setup_process => integration_setup_process

<Integrations: procedures>+≡
    subroutine integration_setup_process (intg, global)
        class(integration_t), intent(inout) :: intg
        type(rt_data_t), intent(inout), target :: global

        class(prc_core_t), allocatable :: core_template
        class(phs_config_t), allocatable :: phs_config_template
        class(mci_t), allocatable :: mci_template
        type(qcd_t) :: qcd
        integer :: n_components, i_component
        type(process_component_def_t), pointer :: config
        type(helicity_selection_t), allocatable :: helicity_selection
        real(default) :: sqrts
        type(sf_config_t), dimension(:), allocatable :: sf_config
        type(sf_channel_t), dimension(:), allocatable :: sf_channel
        class(rng_t), allocatable :: rng
        logical :: rng_seed_known
        integer :: rng_seed
        integer :: n_mci, i_mci

        intg%process => global%process_stack%get_process_ptr (intg%process_id)

!         intg%rng_method = &
!             var_list_get_sval (var_list, var_str ("$rng_method"))

        intg%rebuild_phs = &
            var_list_get_lval (global%var_list, var_str ("?rebuild_phase_space"))
!         intg%check_phs_file = &
!             var_list_get_lval (var_list, var_str ("?check_phs_file"))
!         intg%phs_filename = &
!             var_list_get_sval (var_list, var_str ("$phs_file"))
        intg%phs_only = &
            var_list_get_lval (global%var_list, var_str ("?phs_only"))
!         intg%vis_channels = &
!             var_list_get_lval (var_list, var_str ("?vis_channels"))
!         intg%phs_par%m_threshold_s = &
!             var_list_get_rval (var_list, var_str ("phs_threshold_s"))
!         intg%phs_par%m_threshold_t = &
!             var_list_get_rval (var_list, var_str ("phs_threshold_t"))
!         intg%phs_par%off_shell = &
!             var_list_get_ival (var_list, var_str ("phs_off_shell"))
!         intg%phs_par%keep_nonresonant = &
!             var_list_get_lval (var_list, var_str ("?phs_keep_nonresonant"))
!         intg%phs_par%t_channel = &

```



```

!         var_list_get_ival (var_list, var_str ("phs_t_channel"))
!     intg%mapping_defaults%energy_scale = &
!         var_list_get_rval (var_list, var_str ("phs_e_scale"))
!     intg%mapping_defaults%invariant_mass_scale = &
!         var_list_get_rval (var_list, var_str ("phs_m_scale"))
!     intg%mapping_defaults%momentum_transfer_scale = &
!         var_list_get_rval (var_list, var_str ("phs_q_scale"))
!     intg%mapping_defaults%step_mapping = &
!         var_list_get_lval (var_list, var_str ("?phs_step_mapping"))
!     intg%mapping_defaults%step_mapping_exp = &
!         var_list_get_lval (var_list, var_str ("?phs_step_mapping_exp"))
!     intg%mapping_defaults%enable_s_mapping = &
!         var_list_get_lval (var_list, var_str ("?phs_s_mapping"))

call dispatch_phs (phs_config_template, global)

!     intg%rebuild_grids = &
!         var_list_get_lval (var_list, var_str ("?rebuild_grids"))
!     intg%check_grid_file = &
!         var_list_get_lval (var_list, var_str ("?check_grid_file"))

!     intg%run_id = &
!         var_list_get_sval (var_list, var_str ("$run_id")) ! $
!     if (intg%run_id /= "") then
!         intg%phs_filename_out = intg%process_id // "." // intg%run_id // ".phs"
!         intg%phs_filename_vis = intg%process_id // "." // intg%run_id // "_phs"
!         intg%grids_filename = intg%process_id // "." // intg%run_id // ".vg"
!         intg%history_filename = intg%process_id // "." // intg%run_id &
!             // "-history"
!         intg%log_filename = intg%process_id // "." // intg%run_id // ".log"
!     else
!         intg%phs_filename_out = intg%process_id // ".phs"
!         intg%phs_filename_vis = intg%process_id // "_phs"
!         intg%grids_filename = intg%process_id // ".vg"
!         intg%history_filename = intg%process_id // "-history"
!         intg%log_filename = intg%process_id // ".log"
!     end if
!     intg%use_best_grid = &
!         var_list_get_lval (var_list, var_str ("?use_best_grid"))
!     intg%accuracy_goal = &
!         var_list_get_rval (var_list, var_str ("accuracy_goal"))
!     intg%abs_error_goal = &
!         var_list_get_rval (var_list, var_str ("error_goal"))
!     intg%rel_error_goal = &
!         var_list_get_rval (var_list, var_str ("relative_error_goal"))

call dispatch_mci (mci_template, global, intg%process_id)

!     verb = .true.; if (present (verbose)) verb = verbose
!     if (verb) then
!         call msg_message ("Initializing integration for process " &
!             // char (intg%process_id) // ":")
!         if (intg%run_id /= "") then
!             call msg_message ("Run ID = " // ' ' // char (intg%run_id) // ' ')

```

```

!         end if
!     end if

    helicity_selection = global%get_helicity_selection ()

!     intg%sqrts_known = var_list_is_known (var_list, "sqrts")
!     intg%sqrts = var_list_get_rval (var_list, "sqrts")
!     intg%time_estimate = &
!         var_list_get_lval (var_list, var_str ("?time_estimate"))
!     intg%vis_history = &
!         var_list_get_lval (var_list, var_str ("?vis_history"))
!     intg%n_events_for_me_test = &
!         var_list_get_ival (var_list, var_str ("n_events"))
!     if (associated (global%pn_alpha_qed_expr)) then
!         intg%alpha = eval_real (global%pn_alpha_qed_expr, var_list)
!     else
!         intg%alpha = 0
!     end if
!

    call dispatch_qcd (qcd, global)

    n_components = intg%process%get_n_components ()
    do i_component = 1, n_components
        config => intg%process%get_component_def_ptr (i_component)
        call dispatch_core (core_template, config%get_core_def_ptr (), &
            intg%process%get_model_ptr (), helicity_selection, qcd)
        call intg%process%init_component &
            (i_component, core_template, mci_template, phs_config_template)
    end do

    if (beam_data_are_valid (global%beam_data)) then
        call intg%process%setup_beams (global%beam_data)
    else if (intg%process%get_n_in () == 2) then
        sqrts = var_list_get_rval (global%var_list, var_str ("sqrts"))
        call intg%process%setup_beams (sqrts)
    else
        call msg_bug ("Decay process: Automatic beam setup not supported yet")
!         call intg%process%setup_beams ()
    end if
    call intg%process%beams_startup_message ()

    call dispatch_sf_config (sf_config, sf_channel, global)
    if (size (sf_config) > 0) then
        call intg%process%init_sf_chain (sf_config)
    end if

    call intg%process%configure_phs (intg%rebuild_phs)

    if (allocated (sf_channel)) then
        if (size (sf_channel) > 0) then
            call intg%process%set_sf_channel (sf_channel)
        end if
    end if

```

```

call intg%process%match_channels ()
call intg%process%sf_startup_message ()

call intg%process%setup_mci ()

rng_seed_known = var_list_is_known (global%var_list, var_str ("seed"))
if (rng_seed_known) then
  rng_seed = var_list_get_ival (global%var_list, var_str ("seed"))
end if
n_mci = intg%process%get_n_mci ()
do i_mci = 1, n_mci
  call dispatch_rng (rng, global)
  if (rng_seed_known) then
    call rng%init (rng_seed + i_mci - 1)
  else
    call rng%init ()
  end if
  call intg%process%import_rng (i_mci, rng)
end do

call intg%process%setup_terms ()

call intg%process%set_cuts (global%pn%cuts_lexpr)
call intg%process%set_scale (global%pn%scale_expr)
call intg%process%set_fac_scale (global%pn%fac_scale_expr)
call intg%process%set_ren_scale (global%pn%ren_scale_expr)
call intg%process%set_weight (global%pn%weight_expr)

!   call process_set_alpha_qed (intg%process, intg%alpha)
!   call process_reset_helicity_selection (intg%process, &
!     intg%helicity_selection_threshold, intg%helicity_selection_cutoff)

call intg%process%compute_md5sum ()

end subroutine integration_setup_process

```

Set up beams and structure functions for the process that has been initialized.

*<CCC Integrations: integration: TBP>≡*

```
procedure :: setup_beams => integration_setup_beams
```

*<CCC Integrations: procedures>≡*

```

subroutine integration_setup_beams (intg, sf_list, var_list, ok)
  type(integration_t), intent(inout) :: intg
  type(sf_list_t), pointer :: sf_list
  type(var_list_t), intent(in) :: var_list
  logical, intent(out) :: ok
  if (intg%use_beams) then
    if (beam_data_get_n_in (intg%beam_data) &
      == process_get_n_in (intg%process)) then
      if (associated (sf_list)) then
        intg%sf_list => sf_list
        call process_setup_beams (intg%process, intg%beam_data, &
          sf_list_get_n_strfun (intg%sf_list))
        call process_check_beam_setup (intg%process, var_list)
      end if
    end if
  end if
end subroutine

```

```

        call sf_list_transfer_to_process (intg%sf_list, intg%process)
        intg%use_strfun = .true.
    else
        call process_setup_beams (intg%process, intg%beam_data, 0)
    end if
    ok = .true.
else
    call msg_fatal ("Process '" // char (intg%process_id) &
        // "': beam/process mismatch (collision/decay)", &
        (/ var_str (" -----"), &
        var_str ("This possibly means that you tried to generate "), &
        var_str ("a forbidden process, for which WHIZARD could not"), &
        var_str ("find a matrix element. Or there is a"), &
        var_str ("mismatch between beams and hard interaction.") /) )
    ok = .false.
end if
else if (intg%sqrts_known) then
    call process_setup_beams &
        (intg%process, intg%beam_data, 0, sqrts = intg%sqrts)
else
    call process_setup_beams &
        (intg%process, intg%beam_data, 0)
end if
if (ok) call process_connect_strfun (intg%process, ok)
if (.not. ok) then
    call msg_error ("Process '" // char (intg%process_id) &
        // "': beam/structure function setup failed.")
end if
end subroutine integration_setup_beams

```

Set up the QCD part of the process, i.e., the  $\alpha_s$  handling. This also initializes LHAPDF, if necessary. Data are taken from the structure-function setup, where possible, otherwise from the variable list. As a side-effect, the MD5 checksum for the QCD data is computed and stored in the process data.

*<XXX Integrations: procedures>+≡*

```

subroutine integration_setup_qcd &
    (intg, lhpdf_status, pdf_builtin_status, os_data, var_list)
type(integration_t), intent(inout) :: intg
type(lhpdf_status_t), intent(inout) :: lhpdf_status
type(pdf_builtin_status_t), intent(inout) :: pdf_builtin_status
type(os_data_t), intent(in) :: os_data
type(var_list_t), intent(in) :: var_list
if (intg%alpha_s > 0) call process_set_alpha_s (intg%process, intg%alpha_s)
if (associated (intg%sf_list)) then
    call process_setup_qcd (intg%process, &
        lhpdf_status, pdf_builtin_status, &
        sf_list_get_lhpdf_data_ptr (intg%sf_list), &
        sf_list_get_pdf_builtin_data_ptr (intg%sf_list), &
        os_data, var_list)
else
    call process_setup_qcd (intg%process, &
        lhpdf_status, pdf_builtin_status, &
        null (), null (), &

```

```

        os_data, var_list)
    end if
end subroutine integration_setup_qcd

```

Set up phase space. Must be done after beams setup, since `sqrts` must be known.

```

<CCC Integrations: integration: TBP>+=
  procedure :: setup_phase_space => integration_setup_phase_space

<CCC Integrations: procedures>+=
  subroutine integration_setup_phase_space (intg, os_data, ok)
    type(integration_t), intent(inout) :: intg
    type(os_data_t), intent(in) :: os_data
    logical, intent(out) :: ok
    if (intg%phs_filename == "") then
      call process_setup_phase_space (intg%process, &
        intg%rebuild_phs, &
        os_data, &
        intg%phs_par, intg%mapping_defaults, &
        filename_out = intg%phs_filename_out, &
        filename_vis = intg%phs_filename_vis, &
        vis_channels = intg%vis_channels, &
        check_phs_file = intg%check_phs_file, &
        ok = ok)
    else
      call process_setup_phase_space (intg%process, &
        intg%rebuild_phs, &
        os_data, &
        intg%phs_par, intg%mapping_defaults, &
        filename_in = intg%phs_filename, &
        filename_out = intg%phs_filename_out, &
        filename_vis = intg%phs_filename_vis, &
        vis_channels = intg%vis_channels, &
        ok = ok)
    end if
    if (ok) then
      if (intg%phs_only) then
        call msg_message ("Process '" // char (intg%process_id) &
          // "': phase space setup complete.")
      end if
    else
      call msg_error ("Process '" // char (intg%process_id) &
        // "': phase space setup failed.")
    end if
  end subroutine integration_setup_phase_space

```

The structure function mappings depend on the number of phase-space channels, therefore this comes after the phase-space setup. It completes the structure-function setup.

```

<XXX Integrations: procedures>+=
  subroutine integration_setup_strfun_mappings (intg)
    type(integration_t), intent(inout) :: intg
    if (intg%use_strfun) then

```

```

        call sf_list_setup_mappings (intg%sf_list, intg%process)
    end if
end subroutine integration_setup_strfun_mappings

```

Set up the list of iterations and allocate the VAMP history accordingly.

```

<XXX Integrations: procedures>+≡
subroutine integration_setup_iterations &
    (intg, it_list, it_list_default, ok, verbose)
    type(integration_t), intent(inout) :: intg
    type(iterations_list_t), intent(in) :: it_list
    type(iterations_list_t), dimension(:), intent(in) :: it_list_default
    logical, intent(out) :: ok
    logical, intent(in), optional :: verbose
    integer :: n_in, n_out
    logical :: verb
    verb = .true.; if (present (verbose)) verb = verbose
    ok = .true.
    n_in = process_get_n_in (intg%process)
    n_out = process_get_n_out_real (intg%process)
    intg%it_list = it_list
    if (iterations_list_get_n_pass (intg%it_list) > 0) then
        call iterations_list_complete (intg%it_list, it_list_default(n_out))
    else
        select case (n_out)
        case (:1)
            call msg_error ("Integrate: number of outgoing particles " &
                // "must be at least 2")
            ok = .false.
        case (2:ITERATIONS_DEFAULT_LIST_SIZE)
            intg%it_list = it_list_default(n_out + (n_in-2))
        case default
            intg%it_list = it_list_default(ITERATIONS_DEFAULT_LIST_SIZE)
        end select
    end if
    call iterations_list_adjust_n_calls (intg%it_list, &
        intg%process, intg%grid_parameters)
    if (verb) call iterations_list_write (intg%it_list)
    call process_init_vamp_history &
        (intg%process, iterations_list_get_n_it (intg%it_list))
end subroutine integration_setup_iterations

```

Compute MD5 sums that are used for checking grid files. MD5 sums for cuts etc. are redefined when those structures are set up, see below. Some MD5 sum are computed from process data directly, so they are not set here.

```

<XXX Integrations: procedures>+≡
subroutine integration_collect_md5sums (intg)
    type(integration_t), intent(inout) :: intg
    if (intg%use_beams) then
        intg%md5sum%beams = beam_data_get_md5sum (intg%beam_data, intg%sqrts)
    else
        intg%md5sum%beams = ""
    end if
    if (intg%use_strfun) then

```

```

        intg%md5sum%sf_list = sf_list_get_md5sum (intg%sf_list)
    else
        intg%md5sum%sf_list = ""
    end if
    intg%md5sum%mappings = mapping_defaults_md5sum (intg%mapping_defaults)
end subroutine integration_collect_md5sums

```

Setup up the subevt object inside the process object which is used by cuts, weight, scale.

```

<XXX Integrations: procedures>+≡
subroutine integration_setup_subevt (intg)
    type(integration_t), intent(inout) :: intg
    call process_setup_subevt (intg%process)
end subroutine integration_setup_subevt

```

Set up cuts, user weight, scale.

```

<XXX Integrations: procedures>+≡
subroutine integration_setup_cuts (intg, pn_cuts_lexpr, verbose)
    type(integration_t), intent(inout) :: intg
    type(parse_node_t), pointer :: pn_cuts_lexpr
    logical, intent(in), optional :: verbose
    logical :: verb
    verb = .true.; if (present (verbose)) verb = verbose
    if (associated (pn_cuts_lexpr)) then
        call process_setup_cuts (intg%process, pn_cuts_lexpr, &
            intg%md5sum%cuts)
        if (verb) call msg_message ("Applying user-defined cuts.")
    else
        if (verb) call msg_warning ("No cuts have been defined.")
    end if
end subroutine integration_setup_cuts

```

```

<XXX Integrations: procedures>+≡
subroutine integration_setup_weight (intg, pn_weight_expr, verbose)
    type(integration_t), intent(inout) :: intg
    type(parse_node_t), pointer :: pn_weight_expr
    logical, intent(in), optional :: verbose
    logical :: verb
    verb = .true.; if (present (verbose)) verb = verbose
    if (associated (pn_weight_expr)) then
        call process_setup_weight (intg%process, pn_weight_expr, &
            intg%md5sum%weight)
        if (verb) call msg_message ("Using user-defined reweighting factor.")
    end if
end subroutine integration_setup_weight

```

```

<XXX Integrations: procedures>+≡
subroutine integration_setup_scale (intg, pn_scale_expr, verbose)
    type(integration_t), intent(inout) :: intg
    type(parse_node_t), pointer :: pn_scale_expr
    logical, intent(in), optional :: verbose
    logical :: verb

```

```

verb = .true.; if (present (verbose)) verb = verbose
if (associated (pn_scale_expr)) then
  call process_setup_scale (intg%process, pn_scale_expr, &
    intg%md5sum%scale)
  if (verb) call msg_message ("Using user-defined general scale.")
end if
end subroutine integration_setup_scale

```

*<XXX Integrations: procedures>+≡*

```

subroutine integration_setup_fac_scale (intg, pn_scale_expr, verbose)
  type(integration_t), intent(inout) :: intg
  type(parse_node_t), pointer :: pn_scale_expr
  logical, intent(in), optional :: verbose
  logical :: verb
  verb = .true.; if (present (verbose)) verb = verbose
  if (associated (pn_scale_expr)) then
    call process_setup_fac_scale (intg%process, pn_scale_expr, &
      intg%md5sum%fac_scale)
    if (verb) call msg_message ("Using user-defined factorization scale.")
  end if
end subroutine integration_setup_fac_scale

```

*<XXX Integrations: procedures>+≡*

```

subroutine integration_setup_ren_scale (intg, pn_scale_expr, verbose)
  type(integration_t), intent(inout) :: intg
  type(parse_node_t), pointer :: pn_scale_expr
  logical, intent(in), optional :: verbose
  logical :: verb
  verb = .true.; if (present (verbose)) verb = verbose
  if (associated (pn_scale_expr)) then
    call process_setup_ren_scale (intg%process, pn_scale_expr, &
      intg%md5sum%ren_scale)
    if (verb) call msg_message ("Using user-defined renormalization scale.")
  end if
end subroutine integration_setup_ren_scale

```

Set up integration grids for the first iteration and check whether the grids already contain previous results.

*<XXX Integrations: procedures>+≡*

```

subroutine integration_setup_grids (intg, verbose)
  type(integration_t), intent(inout) :: intg
  logical, intent(in), optional :: verbose
  integer :: n_calls
  logical :: verb, ok
  verb = .true.; if (present (verbose)) verb = verbose
  call process_store_iteration_parameters (intg%process, &
    iterations_list_get_pass_array (intg%it_list), &
    iterations_list_get_n_calls_array (intg%it_list))
  if (iterations_list_get_n_pass (intg%it_list) > 0) then
    n_calls = iterations_list_get_n_calls (intg%it_list, 1)
    if (.not. intg%rebuild_grids) then
      call process_read_grid_file (intg%process, intg%grids_filename, &
        intg%check_grid_file, intg%md5sum, intg%grid_parameters, &

```



```

        iterations_list_get_pass_array (intg%it_list), &
        iterations_list_get_n_calls_array (intg%it_list), &
        ok)
    intg%rebuild_grids = .not. ok
end if
if (intg%rebuild_grids) then
    call process_setup_grids &
        (intg%process, intg%grid_parameters, calls=n_calls)
end if
if (verb) then
    write (msg_buffer, "(4(I0,A),A,L1)" &
        n_calls, " calls, ", &
        process_get_n_channels (intg%process), " channels, ", &
        process_get_n_parameters (intg%process), " dimensions, ", &
        process_get_n_bins (intg%process), " bins, ", &
        "stratified = ", intg%grid_parameters%stratified)
    call msg_message ()
end if
intg%pass_on_file = process_get_current_pass (intg%process)
intg%it_on_file = process_get_current_it (intg%process)
end if
end subroutine integration_setup_grids

```

### 19.9.3 Integration

Write the header for the integration results.

```

<XXX Integrations: procedures>+≡
subroutine integration_write_header (intg, verbose)
    type(integration_t), intent(inout) :: intg
    logical, intent(in), optional :: verbose
    logical :: verb
    integer :: u
    verb = .true.; if (present (verbose)) verb = verbose
    if (verb) then
        call msg_message ("Integrating process '" &
            // char (intg%process_id) // "':")
        u = logfile_unit ()
        call process_results_write_header (intg%process, logfile=.false.)
        if (u > 0) then
            call process_results_write_header (intg%process, unit=u)
            flush (u)
        end if
    end if
end subroutine integration_write_header

```

Integrate: this does a single-iteration pass. If the results are already on file, just write them, otherwise do one VAMP integration per iteration.

```

<XXX Integrations: procedures>+≡
subroutine integration_warmup (intg, rng, pass, verbose)
    type(integration_t), intent(inout) :: intg
    type(tao_random_state), intent(inout) :: rng
    integer, intent(in) :: pass

```

```

logical, intent(in), optional :: verbose
integer :: n_it, n_calls, i, u
logical :: iteration_is_on_file, adapt_grids, adapt_weights
logical :: verb
real(default) :: last_accuracy, current_accuracy
real(default) :: last_abs_error, current_abs_error
real(default) :: last_rel_error, current_rel_error
logical :: accuracy_reached, abs_error_reached, rel_error_reached
logical :: goal_set
verb = .true.; if (present (verbose)) verb = verbose
u = logfile_unit ()
intg%pass = pass
n_calls = iterations_list_get_n_calls (intg%it_list, intg%pass)
if (iterations_list_has_custom_adaptation (intg%it_list, intg%pass)) then
    adapt_grids = iterations_list_adapt_grids (intg%it_list, intg%pass)
    adapt_weights = iterations_list_adapt_weights (intg%it_list, intg%pass)
else
    adapt_grids = .true.
    adapt_weights = .true.
end if
last_accuracy = 0
last_abs_error = 0
last_rel_error = 0
goal_set = intg%accuracy_goal > 0 &
    .or. intg%abs_error_goal > 0 .or. intg%rel_error_goal > 0
accuracy_reached = .false.
abs_error_reached = .false.
rel_error_reached = .false.
n_it = iterations_list_get_n_it (intg%it_list, intg%pass)
LOOP_IT: do i = 1, n_it
    intg%it = intg%it + 1
    iteration_is_on_file = intg%pass < intg%pass_on_file &
        .or. intg%pass == intg%pass_on_file .and. i <= intg%it_on_file
    if (iteration_is_on_file) then
        current_accuracy = process_get_accuracy (intg%process, it=intg%it)
        current_abs_error = process_get_error (intg%process, it=intg%it)
        current_rel_error = process_get_rel_error (intg%process, it=intg%it)
        if (current_accuracy == 0) then
            if (.not. goal_set .or. &
                intg%accuracy_goal > 0 .and. .not. accuracy_reached .or. &
                intg%abs_error_goal > 0 .and. .not. abs_error_reached .or. &
                intg%rel_error_goal > 0 .and. .not. rel_error_reached) then
                call process_discard_results (intg%process, intg%it)
                intg%pass_on_file = process_get_current_pass (intg%process)
                intg%it_on_file = process_get_current_it (intg%process)
                iteration_is_on_file = .false.
            else
                call msg_message &
                    ("Accuracy/error goals reached, skipped iterations")
                intg%it = intg%it + n_it - i
                exit LOOP_IT
            end if
        end if
    end if
end if
end if

```

```

if (iteration_is_on_file) then
  if (verb) then
    call process_results_write_entry (intg%process, intg%it)
    if (u > 0) then
      call process_results_write_entry (intg%process, intg%it, unit=u)
      flush (u)
    end if
  end if
else
  if (.not. goal_set .or. &
    intg%accuracy_goal > 0 .and. .not. accuracy_reached .or. &
    intg%abs_error_goal > 0 .and. .not. abs_error_reached .or. &
    intg%rel_error_goal > 0 .and. .not. rel_error_reached) then
    call process_integrate (intg%process, rng, &
      intg%grid_parameters, &
      intg%pass, 1, 1, n_calls, &
      discard_integrals = i==1, &
      adapt_grids = adapt_grids, &
      adapt_weights = adapt_weights .and. i>2, &
      print_current = verb, &
      time_estimate = intg%time_estimate, &
      grids_filename = intg%grids_filename, &
      write_best_grid = intg%use_best_grid, &
      md5sum = intg%md5sum, &
      history_filename = intg%history_filename, &
      log_filename = intg%log_filename)
  else
    call msg_message &
      ("Accuracy/error goals reached, skipping iterations")
    call process_skip_iterations &
      (intg%process, pass, intg%it - 1, n_it - i + 1)
    intg%it = intg%it + n_it - i
    exit LOOP_IT
  end if
  current_accuracy = process_get_accuracy (intg%process, it=intg%it)
  current_abs_error = process_get_error (intg%process, it=intg%it)
  current_rel_error = process_get_rel_error (intg%process, it=intg%it)
end if
last_accuracy = current_accuracy
last_abs_error = current_abs_error
last_rel_error = current_rel_error
accuracy_reached = last_accuracy < intg%accuracy_goal
abs_error_reached = last_abs_error < intg%abs_error_goal
rel_error_reached = last_rel_error < intg%rel_error_goal
end do LOOP_IT
if (verb) then
  call process_results_write_average (intg%process, intg%pass)
  if (u > 0) then
    call process_results_write_average (intg%process, intg%pass, unit=u)
    flush (u)
  end if
end if
end if
end subroutine integration_warmup

```

Integrate: do the final integration. Here, we do a multi-iteration integration. Again, we skip iterations that are already on file. Record the results in the global variable list.

*(Integrations: integration: TBP)*+≡

```
procedure :: evaluate => integration_evaluate
```

*(Integrations: procedures)*+≡

```
subroutine integration_evaluate (intg, process_instance, i_mci, pass, global)
!   (intg, rng, pass, global_var_list, os_data, verbose)
  class(integration_t), intent(inout) :: intg
  type(process_instance_t), intent(inout), target :: process_instance
!   type(tao_random_state), intent(inout) :: rng
  integer, intent(in) :: i_mci
  integer, intent(in) :: pass
  type(rt_data_t), intent(in), target :: global
!   type(var_list_t), intent(inout) :: global_var_list
!   type(os_data_t), intent(in) :: os_data
!   logical, intent(in), optional :: verbose
!   integer :: it_on_file, u
  integer :: n_calls, n_it
  logical :: adapt_grids, adapt_weights
!   logical :: verb
!   verb = .true.; if (present (verbose)) verb = verbose
!   u = logfile_unit ()
!   intg%pass = pass
!   if (intg%pass == intg%pass_on_file) then
!     it_on_file = intg%it_on_file
!   else
!     it_on_file = 0
!   end if

  associate (it_list => global%it_list)
    if (pass <= it_list%get_n_pass ()) then
      n_calls = it_list%get_n_calls (pass)
      n_it     = it_list%get_n_it (pass)
      adapt_grids = it_list%adapt_grids (pass)
      adapt_weights = it_list%adapt_weights (pass)
    else
      n_calls = intg%process%get_n_calls_default (pass)
      n_it     = intg%process%get_n_it_default (pass)
      adapt_grids = intg%process%adapt_grids_default (pass)
      adapt_weights = intg%process%adapt_weights_default (pass)
    end if
  end associate

!   if (iterations_list_has_custom_adaptation (intg%it_list, intg%pass)) then
!     adapt_grids = iterations_list_adapt_grids (intg%it_list, intg%pass)
!     adapt_weights = iterations_list_adapt_weights (intg%it_list, intg%pass)
!   else
!     adapt_grids = .true.
!     adapt_weights = .false.
!   end if
!   do i = 1, it_on_file
!     intg%it = intg%it + 1
!     if (verb) then
```

```

!         call process_results_write_entry (intg%process, intg%it)
!         if (u > 0) then
!             call process_results_write_entry (intg%process, intg%it, unit=u)
!             flush (u)
!         end if
!     end if
! end do

call intg%process%integrate (process_instance, &
    i_mci, n_it, n_calls, adapt_grids, adapt_weights)

!     call process_integrate (intg%process, rng, &
!         intg%grid_parameters, &
!         intg%pass, it_on_file + 1, n_it, n_calls, &
!         discard_integrals = .true., &
!         adapt_grids = adapt_grids, &
!         adapt_weights = adapt_weights, &
!         print_current = verb, &
!         time_estimate = intg%time_estimate, &
!         grids_filename = intg%grids_filename, &
!         write_best_grid = intg%use_best_grid, &
!         md5sum = intg%md5sum, &
!         history_filename = intg%history_filename, &
!         log_filename = intg%log_filename)
!     if (verb) then
!         call process_results_write_average (intg%process, intg%pass)
!         if (u > 0) then
!             call process_results_write_average (intg%process, intg%pass, unit=u)
!             flush (u)
!         end if
!     end if
!     if (intg%vis_history) then
!         call process_display_integration_history &
!             (intg%process, intg%history_filename, os_data)
!     end if
!     call process_record_integral (intg%process, global_var_list)

end subroutine integration_evaluate

```

Write the footer for the screen display

*<XXX Integrations: procedures>+≡*

```

subroutine integration_write_footer (intg, verbose)
    type(integration_t), intent(in) :: intg
    logical, intent(in), optional :: verbose
    logical :: verb
    integer :: u
    verb = .true.; if (present (verbose)) verb = verbose
    if (verb) then
        u = logfile_unit ()
        call process_results_write_footer (intg%process)
        if (u > 0) then
            call process_results_write_footer (intg%process, unit=u)
            flush (u)
        end if
    end if
end subroutine integration_write_footer

```

```

        end if
        if (intg%time_estimate .and. intg%rebuild_grids) &
            call process_write_time_estimate (intg%process)
        end if
    end subroutine integration_write_footer

```

#### 19.9.4 API for integration objects

This initializer does everything except assigning cuts/scale/weight expressions. If `ok` is false, initialization failed, and the integration should be skipped.

If `me_only` is set, prepare just matrix-element evaluation: no phase space, no iterations.

```

<Integrations: integration: TBP>+≡
    generic :: init => integration_init0 !, integration_init1
    procedure :: integration_init0
    ! procedure :: integration_init1

<Integrations: procedures>+≡
    subroutine integration_init0 (intg, process_id, global)
    !     (intg, process_id, global, ok, me_only, no_beams, verbose)
    class(integration_t), intent(out) :: intg
    type(string_t), intent(in) :: process_id
    type(rt_data_t), intent(inout), target :: global
    !     logical, intent(out) :: ok
    !     logical, intent(in), optional :: me_only, no_beams, verbose
    !     logical :: integrate, allow_beams
    !     integrate = .true.; if (present (me_only)) integrate = .not. me_only
    !     allow_beams = .true.; if (present (no_beams)) allow_beams = .not. no_beams
    call intg%create_process (process_id, global) !, verbose)

    !     if (allow_beams) call integration_check_beam_data (intg, global%beam_data)
    !     call maybe_compile_library (global)

    call intg%setup_process (global)

    !     if (ok) then
    !         call integration_setup_beams &
    !             (intg, global%sf_list, global%var_list, ok)
    !     end if
    !     if (ok) then
    !         call integration_setup_qcd &
    !             (intg, global%lhpdf_status, global%pdf_builtin_status, &
    !             global%os_data, global%var_list)
    !     end if
    !     if (integrate .and. ok) then
    !         call integration_setup_phase_space (intg, global%os_data, ok)
    !     end if
    !     if (integrate .and. ok) then
    !         call integration_setup_strfun_mappings (intg)
    !     end if
    !     if (.not. intg%phs_only) then
    !         if (integrate .and. ok) then
    !             call integration_collect_md5sums (intg)

```

```

!         call integration_setup_iterations &
!           (intg, global%it_list, global%it_list_default, ok, verbose)
!       end if
!       if (ok) then
!         call integration_setup_subevt (intg)
!         call integration_setup_cuts (intg, global%pn_cuts_lexpr, verbose)
!         call integration_setup_scale (intg, global%pn_scale_expr, verbose)
!         call integration_setup_fac_scale (intg, global%pn_fac_scale_expr, verbose)
!         call integration_setup_ren_scale (intg, global%pn_ren_scale_expr, verbose)
!         call integration_setup_weight (intg, global%pn_weight_expr, verbose)
!       end if
!       if (integrate .and. ok) then
!         call integration_setup_grids (intg)
!       end if
!     end if
end subroutine integration_init0

```

Initialize an array of processes.

*(CCC Integrations: procedures)*+≡

```

subroutine integration_init1 &
  (intg, process_id, global, ok, no_beams, me_only, verbose)
  type(integration_t), dimension(:), intent(out) :: intg
  type(string_t), dimension(:), intent(in) :: process_id
  type(rt_data_t), intent(inout), target :: global
  logical, intent(out) :: ok
  logical, intent(in), optional :: no_beams, me_only, verbose
  integer :: proc
  do proc = 1, size (intg)
    call integration_init0 &
      (intg(proc), process_id(proc), global, ok, me_only, verbose)
    if (.not. ok) exit
  end do
end subroutine integration_init1

```

Do the integration for a single process, both warmup and final evaluation.

*(Integrations: integration: TBP)*+≡

```

generic :: integrate => integration_integrate0 !, integration_integrate1
procedure :: integration_integrate0
! procedure :: integration_integrate1

```

*(Integrations: procedures)*+≡

```

subroutine integration_integrate0 (intg, global)
!   (intg, rng, global_var_list, os_data, verbose)
  class(integration_t), intent(inout) :: intg
  type(rt_data_t), intent(inout), target :: global
!   type(tao_random_state), intent(inout) :: rng
!   type(var_list_t), intent(inout) :: global_var_list
!   type(os_data_t), intent(in) :: os_data
!   logical, intent(in), optional :: verbose

  type(process_instance_t), allocatable, target :: process_instance
  integer :: pass, i_mci, n_mci, n_pass

```

```

!      call openmp_set_num_threads_verbose &
!      (var_list_get_ival (global_var_list, var_str ("openmp_num_threads")))
!      call integration_write_header (intg, verbose)

allocate (process_instance)
call process_instance%init (intg%process)

!      do pass = 1, iterations_list_get_n_pass (intg%it_list) - 1
!      call integration_warmup (intg, rng, pass, verbose)
!      end do
!      if (intg%use_best_grid) &
!      call process_choose_best_grid (intg%process, intg%check_grid_file)

n_mci = intg%process%get_n_mci ()
if (n_mci == 1) then
  write (msg_buffer, "(A,A,A)") &
    "Starting integration for process '", &
    char (intg%process%get_id ()), "' "
  call msg_message ()
end if
do i_mci = 1, n_mci
  if (n_mci > 1) then
    write (msg_buffer, "(A,A,A,IO)") &
      "Starting integration for process '", &
      char (intg%process%get_id ()), "' part ", i_mci
    call msg_message ()
  end if
  n_pass = global%it_list%get_n_pass ()
  if (n_pass == 0) then
    call msg_message ("Integrate: iterations not specified, &
      &using default")
    n_pass = intg%process%get_n_pass_default ()
  end if
  do pass = 1, n_pass
    call intg%evaluate (process_instance, i_mci, pass, global)
  end do
  call intg%process%final_integration (i_mci)
end do

!      (intg, rng, pass, global_var_list, os_data, verbose)

!      call integration_write_footer (intg, verbose)

call process_instance%final ()
deallocate (process_instance)

end subroutine integration_integrate0

```

Integrate several processes consecutively.

*(CCC Integrations: procedures)*+≡

```

subroutine integration_integrate1 &
  (intg, rng, global_var_list, os_data, verbose)
  type(integration_t), dimension(:), intent(inout) :: intg

```



```

type(tao_random_state), intent(inout) :: rng
type(var_list_t), intent(inout) :: global_var_list
type(os_data_t), intent(in) :: os_data
logical, intent(in), optional :: verbose
integer :: proc
do proc = 1, size (intg)
    call integration_integrate0 &
        (intg(proc), rng, global_var_list, os_data, verbose)
end do
end subroutine integration_integrate1

```

Do a dummy integration for a process which could not be initialized (e.g., has no matrix element). The result is zero.

*(XXX Integrations: interfaces)*≡

```

interface integration_integrate_dummy
    module procedure integration_integrate_dummy0
    module procedure integration_integrate_dummy1
end interface

```

*(XXX Integrations: procedures)*+≡

```

subroutine integration_integrate_dummy0 (intg, global_var_list, verbose)
    type(integration_t), intent(inout) :: intg
    type(var_list_t), intent(inout) :: global_var_list
    logical, intent(in), optional :: verbose
    call integration_write_header (intg, verbose)
    call process_do_dummy_integration (intg%process)
    call integration_write_footer (intg, verbose)
    call process_record_integral (intg%process, global_var_list)
end subroutine integration_integrate_dummy0

subroutine integration_integrate_dummy1 (intg, global_var_list, verbose)
    type(integration_t), dimension(:), intent(inout) :: intg
    type(var_list_t), intent(inout) :: global_var_list
    logical, intent(in), optional :: verbose
    integer :: proc
    do proc = 1, size (intg)
        call integration_integrate_dummy0 (intg(proc), global_var_list, verbose)
    end do
end subroutine integration_integrate_dummy1

```

Just sample the matrix element under realistic conditions (but no cuts); throw away the results.

*(XXX Integrations: procedures)*+≡

```

subroutine integration_me_test (intg, rng, global_var_list, os_data)
    type(integration_t), intent(inout) :: intg
    type(tao_random_state), intent(inout) :: rng
    type(var_list_t), intent(inout) :: global_var_list
    type(os_data_t), intent(in) :: os_data
    integer :: openmp_num_threads
    real(default) :: time_in_seconds, time_per_call, sample_function_sum
    openmp_num_threads = &
        var_list_get_ival (global_var_list, var_str ("openmp_num_threads"))

```

```

call openmp_set_num_threads_verbose (openmp_num_threads)
write (msg_buffer, "(A,1x,I0,1x,A)") "Matrix element test: " &
// "Calling the sampling function", &
    intg%n_events_for_me_test, "times ..."
call msg_message ()
call process_me_test (intg%process, rng, intg%n_events_for_me_test, &
    time_in_seconds, sample_function_sum)
call msg_message ("... test finished.")
call process_status_write_counters (process_get_status (intg%process))
write (msg_buffer, "(A,1PG22.15)") "Matrix element test: " &
// "Sample function sum:          ", sample_function_sum
call msg_message ()
write (msg_buffer, "(A,1PG12.5)") "Matrix element test: " &
// "Time in seconds (wallclock): ", time_in_seconds
call msg_message ()
if (intg%n_events_for_me_test /= 0) then
    time_per_call = time_in_seconds / intg%n_events_for_me_test
    write (msg_buffer, "(A,1PG12.5)") "Matrix element test: " &
// "Time per call in seconds:    ", time_per_call
    call msg_message ()
    if (openmp_num_threads /= 0) then
        write (msg_buffer, "(A,1PG12.5)") "Matrix element test: " &
// "Time times number of threads:", &
            time_per_call * openmp_num_threads
        call msg_message ()
    end if
end if
end subroutine integration_me_test

```

Prepare the process for matrix-element evaluation, no phase space, no integration.

```

<XXX Integrations: public>≡
    public :: prepare_me_evaluation

<XXX Integrations: interfaces>+≡
    interface prepare_me_evaluation
        module procedure prepare_me_evaluation0
        module procedure prepare_me_evaluation1
    end interface prepare_me_evaluation

<XXX Integrations: procedures>+≡
    subroutine prepare_me_evaluation0 (process_id, global, verbose)
        type(string_t), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global
        logical, intent(in), optional :: verbose
        type(integration_t) :: intg
        logical :: ok
        call integration_init &
            (intg, process_id, global, ok, me_only = .true., verbose = verbose)
    end subroutine prepare_me_evaluation0

    subroutine prepare_me_evaluation1 (process_id, global, verbose)
        type(string_t), dimension(:), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global

```

```

logical, intent(in), optional :: verbose
integer :: proc
do proc = 1, size (process_id)
    call prepare_me_evaluation0 (process_id(proc), global, verbose)
end do
end subroutine prepare_me_evaluation1

```

Prepare the processes that are not yet known.

```

<XXX Integrations: public>+≡
    public :: prepare_me_missing_processes

<XXX Integrations: procedures>+≡
    subroutine prepare_me_missing_processes &
        (process_id, global, verbose)
        type(string_t), dimension(:), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global
        logical, intent(in), optional :: verbose
        integer :: n_proc, n_missing, proc
        type(process_t), pointer :: process
        type(string_t), dimension(:), allocatable :: process_id_missing
        logical, dimension(:), allocatable :: missing
        n_proc = size (process_id)
        allocate (missing (n_proc))
        do proc = 1, n_proc
            process => process_store_get_process_ptr (process_id(proc))
            missing(proc) = .not. associated (process)
        end do
        n_missing = count (missing)
        if (n_missing > 0) then
            allocate (process_id_missing (n_missing))
            process_id_missing = pack (process_id, missing)
            call prepare_me_evaluation (process_id_missing, global, verbose)
        end if
    end subroutine prepare_me_missing_processes

```

Simply integrate, do a dummy integration if necessary. The `integration` object exists only internally.

```

<Integrations: public>≡
    public :: integrate_process

<Integrations: interfaces>≡
    interface integrate_process
        module procedure integrate_process0
    !      module procedure integrate_process1
    end interface

<Integrations: procedures>+≡
    subroutine integrate_process0 (process_id, global)
    !      (process_id, global, global_var_list, no_beams, verbose)
        type(string_t), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global
    !      type(var_list_t), intent(inout) :: global_var_list
    !      logical, intent(in), optional :: no_beams, verbose

```

```

type(string_t) :: prclib_name
type(integration_t) :: intg
!   logical :: ok

if (.not. associated (global%prclib)) then
    call msg_fatal ("Integrate: current process library is undefined")
    return
end if

if (.not. global%prclib%is_active ()) then
    call msg_message ("Integrate: current process library needs compilation")
    prclib_name = global%prclib%get_name ()
    call compile_library (prclib_name, global)
    call msg_message ("Integrate: compilation done")
end if

call intg%init (process_id, global)
!   (intg, process_id, global, ok, no_beams=no_beams, verbose=verbose)

if (intg%phs_only) then
    call msg_message ("Integrate: phase space only, skipping integration")
else
    call intg%integrate (global)
end if

!   if (.not. intg%phs_only) then
!       if (ok) then
!           call integration_integrate &
!               (intg, global%rng, global_var_list, global%os_data, verbose)
!       else
!           call integration_integrate_dummy (intg, global_var_list, verbose)
!       end if
!   end if
end subroutine integrate_process0

!   subroutine integrate_process1 &
!       (process_id, global, global_var_list, no_beams, verbose)
!   type(string_t), dimension(:), intent(in) :: process_id
!   type(rt_data_t), intent(inout), target :: global
!   type(var_list_t), intent(inout) :: global_var_list
!   logical, intent(in), optional :: no_beams, verbose
!   integer :: proc
!   do proc = 1, size (process_id)
!       call integrate_process0 &
!           (process_id(proc), global, global_var_list, no_beams, verbose)
!   end do
! end subroutine integrate_process1
!

```

Do a matrix element test. Prepare for integration, but then just call the matrix element `n_events` number of times.

```

<XXX Integrations: public>+≡
public :: me_test_process

```

```

<XXX Integrations: procedures>+≡
  subroutine me_test_process (process_id, global, global_var_list)
    type(string_t), intent(in) :: process_id
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), intent(inout) :: global_var_list
    type(integration_t) :: intg
    logical :: ok
    call integration_init (intg, process_id, global, ok)
    if (ok) then
      call integration_me_test &
        (intg, global%rng, global_var_list, global%os_data)
    else
      call msg_error ("Matrix element test fails " &
        // "because process has no matrix element")
    end if
  end subroutine me_test_process

```

Integrate the processes that have no integral yet. If no\_beams is set, these include processes that use beams.

```

<XXX Integrations: public>+≡
  public :: integrate_missing_processes

<XXX Integrations: procedures>+≡
  subroutine integrate_missing_processes &
    (process_id, global, global_var_list, no_beams, verbose)
    type(string_t), dimension(:), intent(in) :: process_id
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), intent(inout) :: global_var_list
    logical, intent(in), optional :: no_beams, verbose
    integer :: n_proc, n_missing, proc
    type(process_t), pointer :: process
    type(string_t), dimension(:), allocatable :: process_id_missing
    logical, dimension(:), allocatable :: missing
    type(string_t) :: prc_string
    logical :: verb, nobeams
    verb = .false.; if (present (verbose)) verb = verbose
    nobeams = .false.; if (present (no_beams)) nobeams = no_beams
    n_proc = size (process_id)
    allocate (missing (n_proc))
    do proc = 1, n_proc
      process => process_store_get_process_ptr (process_id(proc))
      if (associated (process)) then
        if (process_has_integral (process)) then
          if (nobeams .and. process_uses_beams (process)) then
            call msg_warning ("Discarding previous result for process '" &
              // char (process_id(proc)) // "': no beam setup allowed")
            missing(proc) = .true.
          else
            missing(proc) = .false.
          end if
        else
          missing(proc) = .false.
        end if
      else
        missing(proc) = .false.
      end if
    end do
  end subroutine integrate_missing_processes

```

```

        missing(proc) = .true.
    end if
end do
n_missing = count (missing)
if (n_missing > 0) then
    allocate (process_id_missing (n_missing))
    process_id_missing = pack (process_id, missing)
    if (verb) then
        prc_string = process_id_missing(1)
        do proc = 2, n_missing
            prc_string = prc_string // ", " // process_id_missing(proc)
        end do
        call msg_message ("Integrating missing processes: " &
            // char (prc_string))
    end if
    if (var_list_get_lval (global%var_list, var_str ("?phs_only"))) then
        call msg_fatal &
            ("Computing missing integrals: ?phs_only must not be set")
    else
        call integrate_process &
            (process_id_missing, global, global_var_list, no_beams, verbose)
    end if
    if (verb) then
        call msg_message ("Integration of missing processes complete.")
    end if
end if
end subroutine integrate_missing_processes

```

### 19.9.5 Test

This is the master for calling self-test procedures.

```

<Integrations: public>+≡
    public :: integrations_test

<Integrations: tests>≡
    subroutine integrations_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
    <Integrations: execute tests>
    end subroutine integrations_test

```

#### Integration of test process

Compile and integrate an intrinsic test matrix element (**prc\_test** type). The phase-space implementation is **phs\_single** (single-particle phase space), the integrator is **mci\_midpoint**.

The cross section for the  $2 \rightarrow 2$  process  $ss \rightarrow ss$  with its constant matrix element is given by

$$\sigma = c \times f \times \Phi_2 \times |M|^2. \quad (19.1)$$

$c$  is the conversion constant

$$c = 0.3894 \times 10^{12} \text{ fb GeV}^2. \quad (19.2)$$

$f$  is the flux of the incoming particles with mass  $m = 125 \text{ GeV}$  and energy  $\sqrt{s} = 1000 \text{ GeV}$

$$f = \frac{(2\pi)^4}{2\lambda^{1/2}(s, m^2, m^2)} = \frac{(2\pi)^4}{2\sqrt{s}\sqrt{s-4m^2}} = 8.048 \times 10^{-4} \text{ GeV}^{-2} \quad (19.3)$$

$\Phi_2$  is the volume of the two-particle phase space

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.5529 \times 10^{-5}. \quad (19.4)$$

The squared matrix element  $|M|^2$  is unity. Combining everything, we obtain

$$\sigma = 8000 \text{ fb} \quad (19.5)$$

This number should appear as the final result.

Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

$\langle \text{Integrations: execute tests} \rangle \equiv$

```
call test (integrations_1, "integrations_1", &
  "intrinsic test process", &
  u, results)
```

$\langle \text{Integrations: tests} \rangle + \equiv$

```
subroutine integrations_1 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global

  write (u, "(A)")  "* Test output: integrations_1"
  write (u, "(A)")  "* Purpose: integrate test process"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()

  libname = "integration_1"
  procname = "prc_config_a"

  call prepare_test_library (global, libname, 1)
  call compile_library (libname, global)

  call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("integrations1"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$_method"), &
    var_str ("unit_test"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
    var_str ("single"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
    var_str ("midpoint"), is_known = .true.)
```

```

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = [ &
    var_str ("method"), &
    var_str ("sqrts"), &
    var_str ("integration_method"), &
    var_str ("phs_method"), &
    var_str ("run_id")])

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_1"

end subroutine integrations_1

```

## Integration with cuts

Compile and integrate an intrinsic test matrix element (prc\_test type) with cuts set.

```

<Integrations: execute tests>+≡
    call test (integrations_2, "integrations_2", &
        "intrinsic test process with cut", &
        u, results)

<Integrations: tests>+≡
    subroutine integrations_2 (u)
        integer, intent(in) :: u
        type(string_t) :: libname, procname
        type(rt_data_t), target :: global

        type(string_t) :: cut_expr_text
        type(ifile_t) :: ifile
        type(stream_t) :: stream
        type(parse_tree_t) :: parse_tree

        type(string_t), dimension(0) :: empty_string_array

        write (u, "(A)")  "* Test output: integrations_2"
        write (u, "(A)")  "* Purpose: integrate test process with cut"
        write (u, "(A)")

        call syntax_model_file_init ()

        call global%global_init ()

```



```

write (u, "(A)")  "* Prepare a cut expression"
write (u, "(A)")

call syntax_pexpr_init ()
cut_expr_text = "all Pt > 100 [s]"
call ifile_append (ifile, cut_expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)
global%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "integration_3"
procname = "prc_config_a"

call prepare_test_library (global, libname, 1)
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("integrations1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
    var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
    var_str ("midpoint"), is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrt_s"), &
    1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = empty_string_array)

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_2"

end subroutine integrations_2

```

## Standard phase space

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the default (`phs_wood`) phase-space implementation. We use an explicit phase-space configuration file with a single channel and integrate by `mci_midpoint`.

```

<Integrations: execute tests>+≡
    call test (integrations_3, "integrations_3", &
        "standard phase space", &
        u, results)

<Integrations: tests>+≡
    subroutine integrations_3 (u)
        integer, intent(in) :: u
        type(string_t) :: libname, procname
        type(rt_data_t), target :: global
        integer :: u_phs

        write (u, "(A)")  "* Test output: integrations_3"
        write (u, "(A)")  "* Purpose: integrate test process"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize process and parameters"
        write (u, "(A)")

        call syntax_model_file_init ()
        call syntax_phs_forest_init ()

        call global%global_init ()

        libname = "integration_3"
        procname = "prc_config_a"

        call prepare_test_library (global, libname, 1)
        call compile_library (libname, global)

        call var_list_set_string (global%var_list, var_str ("$_run_id"), &
            var_str ("integrations1"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$_method"), &
            var_str ("unit_test"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
            var_str ("default"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
            var_str ("midpoint"), is_known = .true.)

        call var_list_set_real (global%var_list, var_str ("sqrts"), &
            1000._default, is_known = .true.)

        write (u, "(A)")  "* Create a scratch phase-space file"
        write (u, "(A)")

        u_phs = free_unit ()
        open (u_phs, file = "integrations_3.phs", &
            status = "replace", action = "write")
        call write_test_phs_file (u_phs, var_str ("prc_config_a_i1"))
        close (u_phs)

        call var_list_set_string (global%var_list, var_str ("$_phs_file"), &
            var_str ("integrations_3.phs"), is_known = .true.)

        call global%it_list%init ([1], [1000])

```

```

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = [ &
    var_str ("phs_method"), &
    var_str ("phs_file")])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_3"

end subroutine integrations_3

```

## VAMP integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the single-channel (`phs_single`) phase-space implementation. The integration method is `vamp`.

```

<Integrations: execute tests>+≡
call test (integrations_4, "integrations_4", &
    "VAMP integration (one iteration)", &
    u, results)

<Integrations: tests>+≡
subroutine integrations_4 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: integrations_4"
    write (u, "(A)")  "* Purpose: integrate test process using VAMP"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and parameters"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    libname = "integrations_4_lib"
    procname = "integrations_4"

    call prepare_test_library (global, libname, 1, [procname])

```

```

call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
    var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = [var_str ("$_integration_method")])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_4"

end subroutine integrations_4

```

## Multiple iterations integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the single-channel (`phs_single`) phase-space implementation. The integration method is `vamp`. We launch three iterations.

```

<Integrations: execute tests>+≡
    call test (integrations_5, "integrations_5", &
        "VAMP integration (three iterations)", &
        u, results)

<Integrations: tests>+≡
subroutine integrations_5 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global

```

```

write (u, "(A)")  "* Test output: integrations_5"
write (u, "(A)")  "* Purpose: integrate test process using VAMP"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and parameters"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()

libname = "integrations_5_lib"
procname = "integrations_5"

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
    var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrt_s"), &
    1000._default, is_known = .true.)

call global%it_list%init ([3], [1000])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = [var_str ("$_integration_method")])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_5"

end subroutine integrations_5

```

## Multiple passes integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the single-channel (`phs_single`) phase-space implementation. The integration method is `vamp`. We launch three passes with three iterations each.

```
<Integrations: execute tests>+≡
    call test (integrations_6, "integrations_6", &
               "VAMP integration (three passes)", &
               u, results)

<Integrations: tests>+≡
    subroutine integrations_6 (u)
        integer, intent(in) :: u
        type(string_t) :: libname, procname
        type(rt_data_t), target :: global
        type(string_t), dimension(0) :: no_vars

        write (u, "(A)")  "* Test output: integrations_6"
        write (u, "(A)")  "* Purpose: integrate test process using VAMP"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize process and parameters"
        write (u, "(A)")

        call syntax_model_file_init ()

        call global%global_init ()

        libname = "integrations_6_lib"
        procname = "integrations_6"

        call prepare_test_library (global, libname, 1, [procname])
        call compile_library (libname, global)

        call var_list_set_string (global%var_list, var_str ("$run_id"), &
                                   var_str ("r1"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$method"), &
                                   var_str ("unit_test"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$phs_method"), &
                                   var_str ("single"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$integration_method"), &
                                   var_str ("vamp"), is_known = .true.)
        call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
                                .false., is_known = .true.)

        call var_list_set_real (global%var_list, var_str ("sqrts"), &
                                1000._default, is_known = .true.)

        call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
                                   adapt = [.true., .true., .false.], &
                                   adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

        write (u, "(A)")  "* Integrate"
        write (u, "(A)")
```

```

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = no_vars)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_6"

end subroutine integrations_6

```

## VAMP and default phase space

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the default (`phs_wood`) phase-space implementation. The integration method is `vamp`. We launch three passes with three iterations each. We enable channel equivalences and groves.

```

<Integrations: execute tests>+≡
call test (integrations_7, "integrations_7", &
          "VAMP integration with wood phase space", &
          u, results)

<Integrations: tests>+≡
subroutine integrations_7 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
  type(string_t), dimension(0) :: no_vars
  integer :: iostat, u_phs
  character(80) :: buffer
  type(string_t) :: phs_file
  logical :: exist

  write (u, "(A)")  "* Test output: integrations_7"
  write (u, "(A)")  "* Purpose: integrate test process using VAMP"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize process and parameters"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()

  libname = "integrations_7_lib"
  procname = "integrations_7"

  call prepare_test_library (global, libname, 1, [procname])

```

```

call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
    var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .true., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
    adapt = [.true., .true., .false.], &
    adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = no_vars)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Generated phase-space file"
write (u, "(A)")

phs_file = procname // "_i1.phs"
inquire (file = char (phs_file), exist = exist)
if (exist) then
    u_phs = free_unit ()
    open (u_phs, file = char (phs_file), action = "read", status = "old")
    iostat = 0
    do while (iostat == 0)
        read (u_phs, "(A)", iostat = iostat)  buffer
        if (iostat == 0) write (u, "(A)")  trim (buffer)
    end do
    close (u_phs)
else
    write (u, "(A)")  "[file is missing]"
end if

write (u, "(A)")

```



```

write (u, "(A)")  "* Test output end: integrations_7"

end subroutine integrations_7

```

## Structure functions

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the default (`phs_wood`) phase-space implementation. The integration method is `vamp`. There is a structure function of type `unit_test`.

We use a test structure function  $f(x) = x$  for both beams. Together with the  $1/x_1 x_2$  factor from the phase-space flux and a unit matrix element, we should get the same result as previously for the process without structure functions. There is a slight correction due to the  $m_s$  mass which we set to zero here.

```

<Integrations: execute tests>+≡
  call test (integrations_8, "integrations_8", &
    "integration with structure function", &
    u, results)

<Integrations: tests>+≡
  subroutine integrations_8 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global
    type(flavor_t) :: flv

    write (u, "(A)")  "* Test output: integrations_8"
    write (u, "(A)")  "* Purpose: integrate test process using VAMP &
      &with structure function"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and parameters"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    libname = "integrations_8_lib"
    procname = "integrations_8"

    call prepare_test_library (global, libname, 1, [procname])
    call compile_library (libname, global)

    call var_list_set_string (global%var_list, var_str ("$_run_id"), &
      var_str ("r1"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("$_method"), &
      var_str ("unit_test"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
      var_str ("wood"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
      var_str ("vamp"), is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &

```

```

        .true., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
    0._default, is_known = .true.)

call reset_interaction_counter ()

call flavor_init (flv, 25, global%model)
call beam_data_init_sqrts (global%beam_data, &
    var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call global%it_list%init ([1], [1000])
call integrate_process (procname, global)

call global%write (u, vars = [var_str ("ms")])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_8"

end subroutine integrations_8

```

## 19.10 Event Streams

This module manages I/O from/to multiple concurrent event streams. Usually, there is at most one input stream, but several output streams. For the latter, we set up an array which can hold `eio_t` (event I/O) objects of different dynamic types simultaneously. One of them may be marked as an input channel.

`<event_streams.f90>`≡  
*<File header>*

```

module event_streams

    <Use kinds>
    <Use strings>
    <Use file utils>
    use diagnostics !NODEP!
    use unit_tests
    use processes

```

```

    use events
    use eio_data
    use eio_base
    use rt_data
    use dispatch

    <Standard module head>

    <Event streams: public>

    <Event streams: parameters>

    <Event streams: types>

contains

    <Event streams: tests>

    <Event streams: procedures>

end module event_streams

```

### 19.10.1 Event Stream Array

Each entry is an `eio_t` object. Since the type is dynamic, we need a wrapper:

```

<Event streams: types>≡
    type :: event_stream_entry_t
        class(eio_t), allocatable :: eio
    end type event_stream_entry_t

```

An array of event-stream entry objects. If one of the entries is an input channel, `i_in` is the corresponding index.

```

<Event streams: public>≡
    public :: event_stream_array_t

<Event streams: types>+≡
    type :: event_stream_array_t
        type(event_stream_entry_t), dimension(:), allocatable :: entry
        integer :: i_in = 0
    contains
        <Event streams: event stream array: TBP>
    end type event_stream_array_t

```

Output.

```

<Event streams: event stream array: TBP>≡
    procedure :: write => event_stream_array_write

<Event streams: procedures>≡
    subroutine event_stream_array_write (object, unit)
        class(event_stream_array_t), intent(in) :: object
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
    end subroutine

```

```

write (u, "(1x,A)") "Event stream array:"
if (allocated (object%entry)) then
  select case (size (object%entry))
  case (0)
    write (u, "(3x,A)") "[empty]"
  case default
    do i = 1, size (object%entry)
      if (i == object%i_in) write (u, "(1x,A)") "Input stream:"
      call object%entry(i)%eio%write (u)
    end do
  end select
else
  write (u, "(3x,A)") "[undefined]"
end if
end subroutine event_stream_array_write

```

Finalize all streams.

```

<Event streams: event stream array: TBP>+≡
  procedure :: final => event_stream_array_final

<Event streams: procedures>+≡
  subroutine event_stream_array_final (es_array)
    class(event_stream_array_t), intent(inout) :: es_array
    integer :: i
    do i = 1, size (es_array%entry)
      call es_array%entry(i)%eio%final ()
    end do
  end subroutine event_stream_array_final

```

Initialization. We use a generic sample name, open event I/O objects for all provides stream types (using the `dispatch_eio` routine), and initialize for the given list of process pointers. If there is an `input` argument, this channel is initialized as an input channel and appended to the array.

```

<Event streams: event stream array: TBP>+≡
  procedure :: init => event_stream_array_init

<Event streams: procedures>+≡
  subroutine event_stream_array_init &
    (es_array, sample, stream_fmt, process_ptr, global, &
     data, input, input_sample, allow_switch, error)
    class(event_stream_array_t), intent(out) :: es_array
    type(string_t), intent(in) :: sample
    type(string_t), dimension(:), intent(in) :: stream_fmt
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(rt_data_t), intent(in) :: global
    type(event_sample_data_t), intent(in), optional :: data
    type(string_t), intent(in), optional :: input
    type(string_t), intent(in), optional :: input_sample
    logical, intent(in), optional :: allow_switch
    logical, intent(out), optional :: error
    type(string_t) :: sample_in
    integer :: n, i
    logical :: success, switch
    if (present (input)) then

```

```

        n = size (stream_fmt) + 1
    else
        n = size (stream_fmt)
    end if
    if (present (input_sample)) then
        sample_in = input_sample
    else
        sample_in = sample
    end if
    if (present (allow_switch)) then
        switch = allow_switch
    else
        switch = .true.
    end if
    if (present (error)) then
        error = .false.
    end if
    allocate (es_array%entry (n))
    if (present (input)) then
        call dispatch_eio (es_array%entry(n)%eio, input, global)
        call es_array%entry(n)%eio%init_in &
            (sample_in, process_ptr, data, success)
        if (success) then
            es_array%i_in = n
        else if (present (input_sample)) then
            if (present (error)) then
                error = .true.
            else
                call msg_fatal ("Events: &
                    &parameter mismatch in input, aborting")
            end if
        else
            call msg_message ("Events: &
                &parameter mismatch, discarding old event set")
            call es_array%entry(n)%eio%final ()
            if (switch) then
                call msg_message ("Events: generating new events")
                call es_array%entry(n)%eio%init_out &
                    (sample, process_ptr, data)
            end if
        end if
    end if
    do i = 1, size (stream_fmt)
        call dispatch_eio (es_array%entry(i)%eio, stream_fmt(i), global)
        call es_array%entry(i)%eio%init_out (sample, process_ptr, data)
    end do
end subroutine event_stream_array_init

```

Switch the (only) input channel to an output channel, so further events are appended to the respective stream.

*(Event streams: event stream array: TBP)*+≡

procedure :: switch\_inout => event\_stream\_array\_switch\_inout

*(Event streams: procedures)*+≡

```

subroutine event_stream_array_switch_inout (es_array)
  class(event_stream_array_t), intent(inout) :: es_array
  integer :: n
  if (es_array%has_input ()) then
    n = es_array%i_in
    call es_array%entry(n)%eio%switch_inout ()
    es_array%i_in = 0
  else
    call msg_bug ("Reading events: switch_inout: no input stream selected")
  end if
end subroutine event_stream_array_switch_inout

```

Output an event (with given process number) to all output streams. If there is no output stream, do nothing.

```

<Event streams: event stream array: TBP>+≡
  procedure :: output => event_stream_array_output

<Event streams: procedures>+≡
  subroutine event_stream_array_output (es_array, event, i_prc)
    class(event_stream_array_t), intent(inout) :: es_array
    type(event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    integer :: i
    do i = 1, size (es_array%entry)
      if (i /= es_array%i_in) then
        call es_array%entry(i)%eio%output (event, i_prc)
      end if
    end do
  end subroutine event_stream_array_output

```

Input the `i_prc` index which selects the process for the current event. This is separated from reading the event, because it determines which event record to read. `iostat` may indicate an error or an EOF condition, as usual.

```

<Event streams: event stream array: TBP>+≡
  procedure :: input_i_prc => event_stream_array_input_i_prc

<Event streams: procedures>+≡
  subroutine event_stream_array_input_i_prc (es_array, i_prc, iostat)
    class(event_stream_array_t), intent(inout) :: es_array
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    integer :: n
    if (es_array%has_input ()) then
      n = es_array%i_in
      call es_array%entry(n)%eio%input_i_prc (i_prc, iostat)
    else
      call msg_fatal ("Reading events: no input stream selected")
    end if
  end subroutine event_stream_array_input_i_prc

```

Input an event from the selected input stream. `iostat` may indicate an error or an EOF condition, as usual.

```

<Event streams: event stream array: TBP>+≡

```

```

    procedure :: input_event => event_stream_array_input_event
  <Event streams: procedures>+≡
    subroutine event_stream_array_input_event (es_array, event, iostat)
      class(event_stream_array_t), intent(inout) :: es_array
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      integer :: n
      if (es_array%has_input ()) then
        n = es_array%i_in
        call es_array%entry(n)%eio%input_event (event, iostat)
      else
        call msg_fatal ("Reading events: no input stream selected")
      end if
    end subroutine event_stream_array_input_event

```

Return true if there is an input channel among the event streams.

```

  <Event streams: event stream array: TBP>+≡
    procedure :: has_input => event_stream_array_has_input

  <Event streams: procedures>+≡
    function event_stream_array_has_input (es_array) result (flag)
      class(event_stream_array_t), intent(in) :: es_array
      logical :: flag
      flag = es_array%i_in /= 0
    end function event_stream_array_has_input

```

## 19.10.2 Unit Tests

```

  <Event streams: public>+≡
    public :: event_streams_test

  <Event streams: tests>≡
    subroutine event_streams_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    <Event streams: execute tests>
    end subroutine event_streams_test

```

### Empty event stream

This should set up an empty event output stream array, including initialization, output, and finalization (which are all no-ops).

```

  <Event streams: execute tests>≡
    call test (event_streams_1, "event_streams_1", &
      "empty event stream array", &
      u, results)

  <Event streams: tests>+≡
    subroutine event_streams_1 (u)
      integer, intent(in) :: u
      type(event_stream_array_t) :: es_array

```

```

type(rt_data_t) :: global
type(event_t) :: event
type(string_t) :: sample
type(string_t), dimension(0) :: empty_string_array
type(process_ptr_t), dimension(0) :: empty_process_ptr_array

write (u, "(A)")  "* Test output: event_streams_1"
write (u, "(A)")  "* Purpose: handle empty event stream array"
write (u, "(A)")

sample = "event_streams_1"

call es_array%init &
    (sample, empty_string_array, empty_process_ptr_array, global)
call es_array%output (event, 42)
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: event_streams_1"

end subroutine event_streams_1

```

## Nontrivial event stream

Here we generate a trivial event and choose raw output as an entry in the stream array.

```

<Event streams: execute tests>+≡
    call test (event_streams_2, "event_streams_2", &
        "nontrivial event stream array", &
        u, results)

<Event streams: tests>+≡
subroutine event_streams_2 (u)
    integer, intent(in) :: u
    type(event_stream_array_t) :: es_array
    type(rt_data_t) :: global
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(process_ptr_t) :: process_ptr
    type(string_t) :: sample
    type(string_t), dimension(0) :: empty_string_array
    integer :: i_prc, iostat

    write (u, "(A)")  "* Test output: event_streams_2"
    write (u, "(A)")  "* Purpose: handle empty event stream array"
    write (u, "(A)")

    write (u, "(A)")  "* Generate test process event"
    write (u, "(A)")

    allocate (process)

```



```

process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance)
call process_instance%setup_event_data ()

allocate (event)
call event%connect (process_instance)
call event%generate (1, [0.4_default, 0.4_default])
call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Allocate raw eio stream and write event to file"
write (u, "(A)")

sample = "event_streams_2"

call es_array%init (sample, [var_str ("raw")], [process_ptr], global)
call es_array%output (event, 1)
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
write (u, "(A)") "* Reallocate raw eio stream for reading"
write (u, "(A)")

sample = "foo"
call es_array%init (sample, empty_string_array, [process_ptr], global, &
    input = var_str ("raw"), input_sample = var_str ("event_streams_2"))
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)

write (u, "(1x,A,I0)") "i_prc = ", i_prc
write (u, "(A)")
call es_array%input_event (event, iostat)
call es_array%final ()

call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Test output end: event_streams_2"

end subroutine event_streams_2

```

### Switch in/out

Here we generate an event file and test switching from writing to reading when the file is exhausted.

```

<Event streams: execute tests>+≡
    call test (event_streams_3, "event_streams_3", &
        "switch input/output", &
        u, results)

<Event streams: tests>+≡
    subroutine event_streams_3 (u)
        integer, intent(in) :: u
        type(event_stream_array_t) :: es_array
        type(rt_data_t) :: global
        type(event_t), allocatable, target :: event
        type(process_t), allocatable, target :: process
        type(process_instance_t), allocatable, target :: process_instance
        type(process_ptr_t) :: process_ptr
        type(string_t) :: sample
        type(string_t), dimension(0) :: empty_string_array
        integer :: i_prc, iostat

        write (u, "(A)")  "* Test output: event_streams_3"
        write (u, "(A)")  "* Purpose: handle in/out switching"
        write (u, "(A)")

        write (u, "(A)")  "* Generate test process event"
        write (u, "(A)")

        allocate (process)
        process_ptr%ptr => process
        allocate (process_instance)
        call prepare_test_process (process, process_instance)
        call process_instance%setup_event_data ()

        allocate (event)
        call event%connect (process_instance)
        call event%generate (1, [0.4_default, 0.4_default])

        write (u, "(A)")  "* Allocate raw eio stream and write event to file"
        write (u, "(A)")

        sample = "event_streams_3"

        call es_array%init (sample, [var_str ("raw")], [process_ptr], global)
        call es_array%output (event, 1)
        call es_array%write (u)
        call es_array%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Reallocate raw eio stream for reading"
        write (u, "(A)")

        call es_array%init (sample, empty_string_array, [process_ptr], global, &
            input = var_str ("raw"))
        call es_array%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Reread event"

```

```

write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)

write (u, "(A)") "* Attempt to read another event (fail), then generate"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
if (iostat < 0) then
    call es_array%switch_inout ()
    call event%generate (1, [0.3_default, 0.3_default])
    call es_array%output (event, 1)
end if
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Reallocate raw eio stream for reading"
write (u, "(A)")

call es_array%init (sample, empty_string_array, [process_ptr], global, &
    input = var_str ("raw"))
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread two events and display 2nd event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)
call es_array%input_i_prc (i_prc, iostat)

call es_array%input_event (event, iostat)
call es_array%final ()

call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Test output end: event_streams_3"

end subroutine event_streams_3

```

## Checksum

Here we generate an event file and repeat twice, once with identical parameters and once with modified parameters.

```

(Event streams: execute tests)+≡
call test (event_streams_4, "event_streams_4", &

```

```

        "check MD5 sum", &
        u, results)
<Event streams: tests>+≡
subroutine event_streams_4 (u)
    integer, intent(in) :: u
    type(event_stream_array_t) :: es_array
    type(rt_data_t) :: global
    type(process_t), allocatable, target :: process
    type(process_ptr_t) :: process_ptr
    type(string_t) :: sample
    type(string_t), dimension(0) :: empty_string_array
    integer :: i_prc, iostat
    type(event_sample_data_t) :: data
    logical :: success

    write (u, "(A)")  "* Test output: event_streams_4"
    write (u, "(A)")  "* Purpose: handle in/out switching"
    write (u, "(A)")

    write (u, "(A)")  "* Generate test process event"
    write (u, "(A)")

    allocate (process)
    process_ptr%ptr => process

    write (u, "(A)")  "* Allocate raw eio stream for writing"
    write (u, "(A)")

    sample = "event_streams_4"
    data%md5sum_cfg = "1234567890abcdef1234567890abcdef"

    call es_array%init &
        (sample, [var_str ("raw")], [process_ptr], global, data)
    call es_array%write (u)
    call es_array%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Reallocate raw eio stream for reading"
    write (u, "(A)")

    call es_array%init (sample, empty_string_array, [process_ptr], global, &
        data, input = var_str ("raw"))
    call es_array%write (u)
    call es_array%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Reallocate modified raw eio stream for reading (fail)"
    write (u, "(A)")

    data%md5sum_cfg = "1234567890_____1234567890_____"
    call es_array%init (sample, empty_string_array, [process_ptr], global, &
        data, input = var_str ("raw"))
    call es_array%write (u)
    call es_array%final ()

```

```

write (u, "(A)")
write (u, "(A)")  "* Test output end: event_streams_4"

end subroutine event_streams_4

```

## 19.11 Simulation

This module manages simulation: event generation and reading/writing of event files. The `simulation` object is intended to be used (via a pointer) outside of WHIZARD, if events are generated individually by an external driver.

```

(simulations.f90)≡
  <File header>

  module simulations

    <Use kinds>
    <Use strings>
    <Use file utils>
    ! use limits, only: MAX_TRIES_FOR_SINGLE_EVENT !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    ! use os_interface
    ! use tao_random_numbers !NODEP!
    ! use cputime
    use sm_qcd
    use md5
    ! use parser
    use variables
    ! use subevents
    ! use expressions
    use flavors
    ! use state_matrices
    use interactions
    use models
    use beams
    use phs_forests
    use rng_base
    use prc_core
    use processes
    ! use decays
    use events
    use eio_data
    use eio_base
    use eio_raw
    use rt_data
    use dispatch
    use process_configurations
    use compilations
    use integrations
    use event_streams

```

```

!    use shower_interface

<Standard module head>

<Simulations: public>

<Simulations: parameters>

<Simulations: types>

contains

<Simulations: tests>

<Simulations: procedures>

end module simulations

```

### 19.11.1 Event counting

In this object we collect statistical information about an event sample or sub-sample.

```

<Simulations: types>≡
  type :: counter_t
    integer :: total = 0
    integer :: generated = 0
    integer :: read = 0
    integer :: positive = 0
    integer :: negative = 0
    integer :: zero = 0
  contains
    <Simulations: counter: TBP>
  end type counter_t

```

Output.

```

<Simulations: counter: TBP>≡
  procedure :: write => counter_write

<Simulations: procedures>≡
  subroutine counter_write (object, unit)
    class(counter_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
1   format (3x,A,I0)
2   format (5x,A,I0)
    write (u, 1) "Events total      = ", object%total
    write (u, 2) "generated        = ", object%generated
    write (u, 2) "read             = ", object%read
    write (u, 2) "positive weight = ", object%positive
    write (u, 2) "negative weight = ", object%negative
    write (u, 2) "zero weight     = ", object%zero
  end subroutine counter_write

```

Count an event. The weight and event source are optional; by default we assume that the event has been generated and has positive weight.

```

<Simulations: counter: TBP>+≡
  procedure :: record => counter_record

<Simulations: procedures>+≡
  subroutine counter_record (counter, weight, from_file)
    class(counter_t), intent(inout) :: counter
    real(default), intent(in), optional :: weight
    logical, intent(in), optional :: from_file
    counter%total = counter%total + 1
    if (present (from_file)) then
      if (from_file) then
        counter%read = counter%read + 1
      else
        counter%generated = counter%generated + 1
      end if
    else
      counter%generated = counter%generated + 1
    end if
    if (present (weight)) then
      if (weight > 0) then
        counter%positive = counter%positive + 1
      else if (weight < 0) then
        counter%negative = counter%negative + 1
      else
        counter%zero = counter%zero + 1
      end if
    else
      counter%positive = counter%positive + 1
    end if
  end subroutine counter_record

```

### 19.11.2 Simulation: component sets

For each set of process components that share a MCI entry in the process configuration, we keep a separate event record.

```

<Simulations: types>+≡
  type :: mci_set_t
    private
    integer :: n_components = 0
    integer, dimension(:), allocatable :: i_component
    type(string_t), dimension(:), allocatable :: component_id
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: weight_mci = 0
    type(counter_t) :: counter
  contains
    <Simulations: mci set: TBP>
  end type mci_set_t

```

Output.

```

<Simulations: mci set: TBP>≡
  procedure :: write => mci_set_write

<Simulations: procedures>+≡
  subroutine mci_set_write (object, unit)
    class(mci_set_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = output_unit (unit)
    write (u, "(3x,A)") "Components:"
    do i = 1, object%n_components
      write (u, "(5x,I0,A,A,A)") object%i_component(i), &
        ": ", char (object%component_id(i)), ""
    end do
    write (u, "(3x,A,ES19.12)") "Integral = ", object%integral
    write (u, "(3x,A,ES19.12)") "Error = ", object%error
    write (u, "(3x,A,F13.10)") "Weight =", object%weight_mci
    call object%counter%write (u)
  end subroutine mci_set_write

```

Initialize: Get the indices and names for the process components that will contribute to this set.

```

<Simulations: mci set: TBP>+≡
  procedure :: init => mci_set_init

<Simulations: procedures>+≡
  subroutine mci_set_init (object, i_mci, process)
    class(mci_set_t), intent(out) :: object
    integer, intent(in) :: i_mci
    type(process_t), intent(in), target :: process
    integer :: i
    call process%get_i_component (i_mci, object%i_component)
    object%n_components = size (object%i_component)
    allocate (object%component_id (object%n_components))
    do i = 1, size (object%component_id)
      object%component_id(i) = &
        process%get_component_id (object%i_component(i))
    end do
    object%integral = process%get_integral (i_mci)
    object%error = process%get_error (i_mci)
  end subroutine mci_set_init

```

Compute the weight for this component set, based on the integral. The `integral_prc` argument should be the sum of component integrals.

```

<Simulations: mci set: TBP>+≡
  procedure :: compute_weight => mci_set_compute_weight

<Simulations: procedures>+≡
  subroutine mci_set_compute_weight (mci_set, integral_prc)
    class(mci_set_t), intent(inout) :: mci_set
    real(default), intent(in) :: integral_prc
    if (integral_prc /= 0) then
      mci_set%weight_mci = mci_set%integral / integral_prc
    end if
  end subroutine mci_set_compute_weight

```



```

        end if
    end subroutine mci_set_compute_weight

```

### 19.11.3 Process-core Safe

This is an object that temporarily holds a process core object. We need this while rescanning a process with modified parameters. After the rescan, we want to restore the original state.

```

<Simulations: types>+≡
    type :: core_safe_t
        class(prc_core_t), allocatable :: core
    end type core_safe_t

```

### 19.11.4 Simulation entry

For each process that we consider for event generation, we need a separate entry. The entry separately records the process ID and run ID. The `weight_mci` array is used for selecting a component set (which shares a MCI record inside the process container) when generating an event for the current process.

The simulation entry is an extension of the `event_t` event record. This core object contains configuration data, pointers to the process and process instance, the expressions, flags and values that are evaluated at runtime, and the resulting particle set.

The entry explicitly allocate the `process_instance`, which becomes the process-specific workspace for the event record.

```

<Simulations: types>+≡
    type, extends (event_t) :: entry_t
    private
        type(string_t) :: process_id
        type(string_t) :: library
        type(string_t) :: run_id
        real(default) :: integral = 0
        real(default) :: error = 0
        real(default) :: weight_prc = 0
        type(counter_t) :: counter
        integer :: n_mci = 0
        type(mci_set_t), dimension(:), allocatable :: mci_set
        real(default), dimension(:), allocatable :: acc_weight_mci
        type(core_safe_t), dimension(:), allocatable :: core_safe
    contains
        <Simulations: entry: TBP>
    end type entry_t

```

Output. Write just the configuration, the event is written by a separate routine.

The `verbose` option is unused, it is required by the interface of the base-object method.

```

<Simulations: entry: TBP>≡
    procedure :: write => entry_write

```

*<Simulations: procedures>+≡*

```

subroutine entry_write (object, unit, verbose)
  class(entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i
  u = output_unit (unit)
  write (u, "(3x,A,A,A)") "Process   = ', char (object%process_id), "'
  write (u, "(3x,A,A,A)") "Library   = ', char (object%library), "'
  write (u, "(3x,A,A,A)") "Run       = ', char (object%run_id), "'
  write (u, "(3x,A,ES19.12)") "Integral = ", object%integral
  write (u, "(3x,A,ES19.12)") "Error    = ", object%error
  write (u, "(3x,A,F13.10)") "Weight   =", object%weight_prc
  write (u, "(3x,A,I0)") "MCI sets = ", object%n_mci
  call object%counter%write (u)
  do i = 1, size (object%mci_set)
    write (u, "(A)")
    write (u, "(1x,A,I0,A)") "MCI set #", i, ":"
    call object%mci_set(i)%write (u)
  end do
  if (allocated (object%core_safe)) then
    do i = 1, size (object%core_safe)
      write (u, "(1x,A,I0,A)") "Saved process-component core #", i, ":"
      call object%core_safe(i)%core%write (u)
    end do
  end if
end subroutine entry_write

```

Write the event and process instance to an (optional) output unit.

*<Simulations: entry: TBP>+≡*

```

procedure :: write_event => entry_write_event

```

*<Simulations: procedures>+≡*

```

subroutine entry_write_event (object, unit, verbose)
  class(entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  call object%event_t%write (unit, verbose)
end subroutine entry_write_event

```

Finalizer. The `instance` pointer component of the `event_t` base type points to a target which we did explicitly allocate in the `entry_init` procedure. Therefore, we finalize and explicitly deallocate it here. Then we call the finalizer of the base type.

*<Simulations: entry: TBP>+≡*

```

procedure :: final => entry_final

```

*<Simulations: procedures>+≡*

```

subroutine entry_final (object)
  class(entry_t), intent(inout) :: object
  integer :: i
  if (associated (object%instance)) then
    do i = 1, object%n_mci
      call object%process%final_simulation (object%instance, i)
    end do
  end if
end subroutine entry_final

```

```

        end do
        call object%instance%final ()
        deallocate (object%instance)
    end if
    call object%event_t%final ()
end subroutine entry_final

```

Initialization. Search for a process entry and allocate a process instance as an anonymous object, temporarily accessible via the `process_instance` pointer. Assign data by looking at the process object and at the environment.

If the process object is not found initially, attempt an integration pass and try again.

When done, we assign the `instance` and `process` pointers of the base type by the `connect` method, so we can reference them later.

```

<Simulations: entry: TBP>+≡
    procedure :: init => entry_init

<Simulations: procedures>+≡
    subroutine entry_init (entry, process_id, global)
        class(entry_t), intent(inout), target :: entry
        type(string_t), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global
        type(process_t), pointer :: process
        type(process_instance_t), pointer :: process_instance
        integer :: i
        process => global%process_stack%get_process_ptr (process_id)
        if (.not. associated (process)) then
            call msg_message ("Simulate: process '" &
                // char (process_id) // "' needs integration")
            call integrate_process (process_id, global)
            process => global%process_stack%get_process_ptr (process_id)
            if (associated (process)) then
                call msg_message ("Simulate: integration done")
            else
                call msg_fatal ("Simulate: process '" &
                    // char (process_id) // "' could not be integrated: aborting")
            end if
        end if
        call entry%basic_init (global%var_list)
        allocate (process_instance)
        call process_instance%init (process)
        call process_instance%setup_event_data ()
        entry%process_id = process_id
        entry%library = process%get_library_name ()
        entry%run_id = process%get_run_id ()
        entry%n_mci = process%get_n_mci ()
        allocate (entry%mci_set (entry%n_mci))
        do i = 1, size (entry%mci_set)
            call entry%mci_set(i)%init (i, process)
        end do
        entry%integral = process%get_integral ()
        entry%error = process%get_error ()
        call entry%set_selection (global%pn%selection_lexpr)
    end subroutine entry_init

```

```

call entry%set_reweight (global%pn%reweight_expr)
call entry%set_analysis (global%pn%analysis_lexpr)
do i = 1, entry%n_mci
    call process%init_simulation (process_instance, i)
end do
call entry%connect (process_instance)
call entry%setup_expressions ()
end subroutine entry_init

```

Compute weights. The integral in the argument is the sum of integrals for all processes in the sample. After computing the process weights, we repeat the normalization procedure for the process components.

*<Simulations: entry: TBP>+≡*

```

procedure :: compute_weights => entry_compute_weights

```

*<Simulations: procedures>+≡*

```

subroutine entry_compute_weights (entry, integral_sample)
class(entry_t), intent(inout) :: entry
real(default), intent(in) :: integral_sample
integer :: i
if (integral_sample /= 0) then
    entry%weight_prc = entry%integral / integral_sample
end if
allocate (entry%acc_weight_mci (entry%n_mci))
do i = 1, entry%n_mci
    call entry%mciset(i)%compute_weight (entry%integral)
    if (i == 1) then
        entry%acc_weight_mci(i) = entry%mciset(i)%weight_mci
    else if (i < entry%n_mci) then
        entry%acc_weight_mci(i) = entry%mciset(i)%weight_mci &
            + entry%acc_weight_mci(i-1)
    else
        entry%acc_weight_mci(i) = 1
    end if
end do
end subroutine entry_compute_weights

```

Select a MCI entry, using the random-number generator. If there is only one component, we may go along with an unallocated generator.

*<Simulations: entry: TBP>+≡*

```

procedure :: select_mci => entry_select_mci

```

*<Simulations: procedures>+≡*

```

function entry_select_mci (entry, rng) result (i_mci)
class(entry_t), intent(in) :: entry
class(rng_t), intent(inout) :: rng
integer :: i_mci
integer :: i
real(default) :: x
select case (entry%n_mci)
case (0)
    i_mci = 0
case (1)
    i_mci = 1

```

```

case default
  call rng%generate (x)
  do i = 1, entry%n_mci
    if (x <= entry%acc_weight_mci(i)) then
      i_mci = i
      exit
    end if
  end do
end select
end function entry_select_mci

```

Record an event for this entry, i.e., increment the appropriate counters.

```

⟨Simulations: entry: TBP⟩+≡
  procedure :: record => entry_record

⟨Simulations: procedures⟩+≡
  subroutine entry_record (entry, i_mci, from_file)
    class(entry_t), intent(inout) :: entry
    integer, intent(in) :: i_mci
    logical, intent(in), optional :: from_file
    real(default) :: weight
    weight = entry%get_weight ()
    call entry%counter%record (weight, from_file)
    call entry%mci_set(i_mci)%counter%record (weight)
  end subroutine entry_record

```

Rescale the event weight for this entry by the ratio of the process instance (recalculated) sqme and the event sqme.

```

⟨Simulations: entry: TBP⟩+≡
  procedure :: rescale_weight => entry_rescale_weight

⟨Simulations: procedures⟩+≡
  subroutine entry_rescale_weight (entry)
    class(entry_t), intent(inout) :: entry
    real(default) :: sqme_evt, sqme_prc, weight
    sqme_evt = entry%get_sqme ()
    weight = entry%get_weight ()
    sqme_prc = entry%get_sqme_process ()
    if (sqme_evt /= 0) then
      call entry%set_sqme (sqme_evt, weight * sqme_prc / sqme_evt)
    else
      call entry%set_sqme (0._default, 0._default)
    end if
  end subroutine entry_rescale_weight

```

Update and restore the process core that this entry accesses, when parameters change.

```

⟨Simulations: entry: TBP⟩+≡
  procedure :: update_process => entry_update_process
  procedure :: restore_process => entry_restore_process

```

*<Simulations: procedures>+≡*

```

subroutine entry_update_process (entry, model, helicity_selection, qcd)
  class(entry_t), intent(inout) :: entry
  type(model_t), intent(in), optional, target :: model
  type(helicity_selection_t), intent(in), optional :: helicity_selection
  type(qcd_t), intent(in), optional :: qcd
  type(process_t), pointer :: process
  class(prc_core_t), allocatable :: core
  integer :: i, n_components
  process => entry%get_process_ptr ()
  n_components = process%get_n_components ()
  allocate (entry%core_safe (n_components))
  do i = 1, n_components
    call process%extract_component_core (i, core)
    call dispatch_core_update (core, model, helicity_selection, qcd, &
      entry%core_safe(i)%core)
    call process%restore_component_core (i, core)
  end do
end subroutine entry_update_process

subroutine entry_restore_process (entry)
  class(entry_t), intent(inout) :: entry
  type(process_t), pointer :: process
  class(prc_core_t), allocatable :: core
  integer :: i, n_components
  process => entry%get_process_ptr ()
  n_components = process%get_n_components ()
  do i = 1, n_components
    call process%extract_component_core (i, core)
    call dispatch_core_restore (core, entry%core_safe(i)%core)
    call process%restore_component_core (i, core)
  end do
  deallocate (entry%core_safe)
end subroutine entry_restore_process

```

### 19.11.5 The simulation type

Each simulation object corresponds to an event sample, identified by the `sample_id`.

The simulation may cover several processes simultaneously. All process-specific data, including the event records, are stored in the `entry` subobjects. The `current` index indicates which record was selected last.

*<Simulations: public>≡*

```
public :: simulation_t
```

*<Simulations: types>+≡*

```

type :: simulation_t
  private
  type(string_t) :: sample_id
  logical :: unweighted = .true.
  logical :: negative_weights = .false.
  integer :: n_prc = 0
  real(default) :: integral = 0
  real(default) :: error = 0

```

```

character(32) :: md5sum_prc = ""
character(32) :: md5sum_cfg = ""
type(entry_t), dimension(:), allocatable :: entry
real(default), dimension(:), allocatable :: acc_weight_prc
integer :: n_evt_requested = 0
type(counter_t) :: counter
class(rng_t), allocatable :: rng
integer :: i_prc = 0
integer :: i_mci = 0
real(default) :: weight = 0
contains
  <Simulations: simulation: TBP>
end type simulation_t

```

Output. `write` writes just the configuration. `write_event` writes the current event (and process instance).

```

<Simulations: simulation: TBP>≡
  procedure :: write => simulation_write

<Simulations: procedures>+≡
  subroutine simulation_write (object, unit, verbose)
    class(simulation_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u, i
    u = output_unit (unit)
    call write_separator_double (u)
    write (u, "(1x,A,A,A)") "Event sample: '", char (object%sample_id), "'
    write (u, "(3x,A,I0)") "Processes = ", object%n_prc
    write (u, "(3x,A,L1)") "Unweighted = ", object%unweighted
    !   write (u, "(3x,A,L1)") "Neg.weights = ", object%negative_weights
    write (u, "(3x,A,ES19.12)") "Integral = ", object%integral
    write (u, "(3x,A,ES19.12)") "Error = ", object%error
    if (object%md5sum_prc /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (proc) = '", object%md5sum_prc, "'
    end if
    if (object%md5sum_cfg /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (config) = '", object%md5sum_cfg, "'
    end if
    write (u, "(3x,A,I0)") "Events requested = ", object%n_evt_requested
    call object%counter%write (u)
    call write_separator (u)
    if (object%i_prc /= 0) then
      write (u, "(1x,A)") "Current event:"
      write (u, "(3x,A,I0,A,A)") "Process #", &
        object%i_prc, ": ", &
        char (object%entry(object%i_prc)%process_id)
      write (u, "(3x,A,I0)") "MCI set #", object%i_mci
      write (u, "(3x,A,ES19.12)") "Weight = ", object%weight
    else
      write (u, "(1x,A,I0,A,A)") "Current event: [undefined]"
    end if
    call write_separator (u)
    if (allocated (object%rng)) then

```

```

        call object%rng%write (u)
    else
        write (u, "(3x,A)") "Random-number generator: [undefined]"
    end if
    if (allocated (object%entry)) then
        do i = 1, size (object%entry)
            if (i == 1) then
                call write_separator_double (u)
            else
                call write_separator (u)
            end if
            write (u, "(1x,A,I0,A)") "Process #", i, ":"
            call object%entry(i)%write (u, verbose)
        end do
    end if
    call write_separator_double (u)
end subroutine simulation_write

```

Write the current event record. If an explicit index is given, write that event record.

We implement writing to `unit` (event contents / debugging format) and writing to an `eio` event stream (storage).

```

<Simulations: simulation: TBP>+≡
    generic :: write_event => write_event_unit
    procedure :: write_event_unit => simulation_write_event_unit

<Simulations: procedures>+≡
    subroutine simulation_write_event_unit (object, unit, i_prc, verbose)
        class(simulation_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
        integer, intent(in), optional :: i_prc
        integer :: current
        if (present (i_prc)) then
            current = i_prc
        else
            current = object%i_prc
        end if
        if (current > 0) then
            call object%entry(current)%write_event (unit, verbose)
        else
            call msg_fatal ("Simulation: write event: no process selected")
        end if
    end subroutine simulation_write_event_unit

```

Finalizer.

```

<Simulations: simulation: TBP>+≡
    procedure :: final => simulation_final

<Simulations: procedures>+≡
    subroutine simulation_final (object)
        class(simulation_t), intent(inout) :: object
        integer :: i
        do i = 1, size (object%entry)

```



```

        call object%entry(i)%final ()
    end do
end subroutine simulation_final

```

Initialization. We can deduce all data from the given list of process IDs and the global data set. The process objects are taken from the stack. Once the individual integrals are known, we add them (and the errors), to get the sample integral.

```

<Simulations: simulation: TBP>+=
    procedure :: init => simulation_init

<Simulations: procedures>+=
    subroutine simulation_init (simulation, process_id, global)
        class(simulation_t), intent(out), target :: simulation
        type(string_t), dimension(:), intent(in) :: process_id
        type(rt_data_t), intent(inout), target :: global
        logical :: rng_seed_known
        integer :: rng_seed
        integer :: i
        simulation%sample_id = var_list_get_sval (global%var_list, &
            var_str ("sample"))
        simulation%unweighted = var_list_get_lval (global%var_list, &
            var_str ("unweighted"))
        ! simulation%negative_weights = var_list_get_lval (global%var_list, &
        ! var_str ("negative_weights"))
        select case (size (process_id))
        case (0)
            call msg_error ("Simulation: no process selected")
        case (1)
            write (msg_buffer, "(A,A,A)") &
                "Starting simulation for process '", &
                char (process_id(1)), "'"
            call msg_message ()
        case default
            write (msg_buffer, "(A,A,A)") &
                "Starting simulation for processes '", &
                char (process_id(1)), "' etc."
            call msg_message ()
        end select
        simulation%n_prc = size (process_id)
        allocate (simulation%entry (simulation%n_prc))
        do i = 1, size (simulation%entry)
            call simulation%entry(i)%init (process_id(i), global)
        end do
        call dispatch_rng (simulation%rng, global)
        rng_seed_known = var_list_is_known (global%var_list, var_str ("seed"))
        if (rng_seed_known) then
            rng_seed = var_list_get_ival (global%var_list, var_str ("seed"))
            call simulation%rng%init (rng_seed)
        else
            call simulation%rng%init ()
        end if
        simulation%integral = sum (simulation%entry%integral)
        simulation%error = sqrt (sum (simulation%entry%error ** 2))
    end subroutine simulation_init

```

```

        call simulation%compute_md5sum ()
    end subroutine simulation_init

```

Compute the checksum of the process set. We retrieve the MD5 sums of all processes. This depends only on the process definitions, while parameters are not considered. The configuration checksum is retrieved from the MCI records in the process objects and furthermore includes beams, parameters, integration results, etc., so matching the latter should guarantee identical physics.

```

<Simulations: simulation: TBP>+≡
    procedure :: compute_md5sum => simulation_compute_md5sum

<Simulations: procedures>+≡
    subroutine simulation_compute_md5sum (simulation)
        class(simulation_t), intent(inout) :: simulation
        type(process_t), pointer :: process
        type(string_t) :: buffer
        integer :: i, n_mci, i_mci, n_component, i_component
        if (simulation%md5sum_prc == "") then
            buffer = ""
            do i = 1, simulation%n_prc
                process => simulation%entry(i)%get_process_ptr ()
                n_component = process%get_n_components ()
                do i_component = 1, n_component
                    buffer = buffer // process%get_md5sum_prc (i_component)
                end do
            end do
            simulation%md5sum_prc = md5sum (char (buffer))
        end if
        if (simulation%md5sum_cfg == "") then
            buffer = ""
            do i = 1, simulation%n_prc
                process => simulation%entry(i)%get_process_ptr ()
                n_mci = process%get_n_mci ()
                do i_mci = 1, n_mci
                    buffer = buffer // process%get_md5sum_mci (i_mci)
                end do
            end do
            simulation%md5sum_cfg = md5sum (char (buffer))
        end if
    end subroutine simulation_compute_md5sum

```

Compute weights: normalize the integrals for the processes by the total sum.

```

<Simulations: simulation: TBP>+≡
    procedure :: compute_weights => simulation_compute_weights

<Simulations: procedures>+≡
    subroutine simulation_compute_weights (simulation)
        class(simulation_t), intent(inout) :: simulation
        integer :: i
        allocate (simulation%acc_weight_prc (simulation%n_prc))
        do i = 1, simulation%n_prc
            call simulation%entry(i)%compute_weights (simulation%integral)
            if (i == 1) then

```

```

        simulation%acc_weight_prc(i) = simulation%entry(i)%weight_prc
    else if (i < simulation%n_prc) then
        simulation%acc_weight_prc(i) = simulation%entry(i)%weight_prc &
            + simulation%acc_weight_prc(i-1)
    else
        simulation%acc_weight_prc(i) = 1
    end if
end do
end subroutine simulation_compute_weights

```

Select a process, using the random-number generator.

*<Simulations: simulation: TBP>+≡*

```

    procedure :: select_prc => simulation_select_prc

```

*<Simulations: procedures>+≡*

```

    function simulation_select_prc (simulation) result (i_prc)
        class(simulation_t), intent(inout) :: simulation
        integer :: i_prc
        integer :: i
        real(default) :: x
        select case (simulation%n_prc)
        case (0)
            i_prc = 0
        case (1)
            i_prc = 1
        case default
            call simulation%rng%generate (x)
            do i = 1, simulation%n_prc
                if (x <= simulation%acc_weight_prc(i)) then
                    i_prc = i
                    exit
                end if
            end do
        end select
    end function simulation_select_prc

```

Select a MCI set for the selected process.

*<Simulations: simulation: TBP>+≡*

```

    procedure :: select_mci => simulation_select_mci

```

*<Simulations: procedures>+≡*

```

    function simulation_select_mci (simulation) result (i_mci)
        class(simulation_t), intent(inout) :: simulation
        integer :: i_mci
        if (simulation%i_prc /= 0) then
            i_mci = simulation%entry(simulation%i_prc)%select_mci (simulation%rng)
        end if
    end function simulation_select_mci

```

Generate a predefined number of events. First select a process and a component set, then generate an event for that process and factorize the quantum state. The pair of random numbers can be used for factorization.

*<Simulations: simulation: TBP>+≡*

```

    procedure :: generate => simulation_generate

```

*<Simulations: procedures>+≡*

```

subroutine simulation_generate (simulation, n, es_array)
  class(simulation_t), intent(inout) :: simulation
  integer, intent(in) :: n
  type(event_stream_array_t), intent(inout), optional :: es_array
  real(default), dimension(2) :: r
  type(string_t) :: str1, str2
  logical :: generate_new
  integer :: i
  simulation%n_evt_requested = n
  str1 = "Generating"
  if (present (es_array)) then
    if (es_array%has_input ()) str1 = "Reading"
  end if
  if (simulation%entry(1)%config%unweighted) then
    str2 = "unweighted"
  else
    str2 = "weighted"
  end if
  write (msg_buffer, "(A,1x,I0,1x,A,1x,A)") char (str1), n, char (str2), &
    "events ..."
  call msg_message ()
  do i = 1, n
    if (present (es_array)) then
      call simulation%read_event (es_array, .true., generate_new)
    else
      generate_new = .true.
    end if
    if (generate_new) then
      simulation%i_prc = simulation%select_prc ()
      simulation%i_mci = simulation%select_mci ()
      call simulation%rng%generate (r)
      associate (entry => simulation%entry(simulation%i_prc))
        call entry%generate (simulation%i_mci, r)
        simulation%weight = entry%get_weight ()
        call simulation%counter%record (simulation%weight)
        call entry%record (simulation%i_mci)
      end associate
    else
      associate (entry => simulation%entry(simulation%i_prc))
        simulation%weight = entry%get_weight ()
        call simulation%counter%record (simulation%weight, from_file=.true.)
        call entry%record (simulation%i_mci, from_file=.true.)
      end associate
    end if
    if (present (es_array)) then
      call simulation%write_event (es_array)
    end if
  end do
  call msg_message ("... event sample complete.")
end subroutine simulation_generate

```

Rescan an undefined number of events.

If `update_event` or `update_sqme` is set, we have to recalculate the event,

starting from the particle set. If the latter is set, this includes the squared matrix element (i.e., the amplitude is evaluated). Otherwise, only kinematics and observables derived from it are recovered.

Not everything is updated. Missing: helicity selection data, PDF, etc.

```

<Simulations: simulation: TBP>+≡
  procedure :: rescan => simulation_rescan

<Simulations: procedures>+≡
  subroutine simulation_rescan &
    (simulation, es_array, update_event, update_sqme, update_weight, &
     model, helicity_selection, qcd)
    class(simulation_t), intent(inout) :: simulation
    type(event_stream_array_t), intent(inout) :: es_array
    logical, intent(in) :: update_event, update_sqme, update_weight
    type(model_t), intent(in), optional, target :: model
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    type(qcd_t), intent(in), optional :: qcd
    real(default), dimension(2) :: r
    type(string_t) :: str1, str2, str3
    logical :: complete
    simulation%n_evt_requested = 0
    str1 = "Rescanning"
    if (simulation%entry(1)%config%unweighted) then
      str2 = "unweighted"
    else
      str2 = "weighted"
    end if
    if (update_sqme .or. update_weight) then
      str3 = "(process parameters updated) "
      call simulation%update_processes (model, helicity_selection, qcd)
    else
      str3 = ""
    end if
    write (msg_buffer, "(A,1x,A,1x,A,A,A)") char (str1), char (str2), &
      "events ", char (str3), "..."
    call msg_message ()
    do
      call simulation%read_event (es_array, .false., complete)
      if (complete) exit
      if (update_event .or. update_sqme .or. update_weight) then
        call simulation%recalculate (update_sqme)
        if (update_weight) call simulation%rescale_weight ()
      end if
      associate (entry => simulation%entry(simulation%i_prc))
        simulation%weight = entry%get_weight ()
        call simulation%counter%record (simulation%weight, from_file=.true.)
        call entry%record (simulation%i_mci, from_file=.true.)
      end associate
      call simulation%write_event (es_array)
    end do
    if (update_sqme .or. update_weight) then
      call simulation%restore_processes ()
    end if
  end subroutine simulation_rescan

```

Rescale the current event weight: we multiply the event weight by the ratio of the `sqme` stored in the process instance (presumably recalculated) over the `sqme` stored in the event record (presumably read from file).

```

<Simulations: simulation: TBP>+=
  procedure :: rescale_weight => simulation_rescale_weight

<Simulations: procedures>+=
  subroutine simulation_rescale_weight (simulation)
    class(simulation_t), intent(inout) :: simulation
    call simulation%entry(simulation%i_prc)%rescale_weight ()
  end subroutine simulation_rescale_weight

```

These routines take care of temporary parameter redefinitions that we want to take effect while recalculating the matrix elements. We extract the core(s) of the processes that we are simulating, apply the changes, and make sure that the changes are actually used. This is the duty of `dispatch_core_update`. When done, we restore the original versions using `dispatch_core_restore`.

```

<Simulations: simulation: TBP>+=
  procedure :: update_processes => simulation_update_processes
  procedure :: restore_processes => simulation_restore_processes

<Simulations: procedures>+=
  subroutine simulation_update_processes &
    (simulation, model, helicity_selection, qcd)
    class(simulation_t), intent(inout) :: simulation
    type(model_t), intent(in), optional, target :: model
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    type(qcd_t), intent(in), optional :: qcd
    integer :: i
    do i = 1, simulation%n_prc
      call simulation%entry(i)%update_process (model, helicity_selection, qcd)
    end do
  end subroutine simulation_update_processes

  subroutine simulation_restore_processes (simulation)
    class(simulation_t), intent(inout) :: simulation
    integer :: i
    do i = 1, simulation%n_prc
      call simulation%entry(i)%restore_process ()
    end do
  end subroutine simulation_restore_processes

```

### 19.11.6 Event Stream I/O

Write an event to a generic `eio` event stream. The process index must be selected, or the current index must be available.

```

<Simulations: simulation: TBP>+=
  generic :: write_event => write_event_eio
  procedure :: write_event_eio => simulation_write_event_eio

```

*<Simulations: procedures>+≡*

```

subroutine simulation_write_event_eio (object, eio, i_prc)
  class(simulation_t), intent(in) :: object
  class(eio_t), intent(inout) :: eio
  integer, intent(in), optional :: i_prc
  integer :: current
  if (present (i_prc)) then
    current = i_prc
  else
    current = object%i_prc
  end if
  if (current > 0) then
    call eio%output (object%entry(current)%event_t, current)
  else
    call msg_fatal ("Simulation: write event: no process selected")
  end if
end subroutine simulation_write_event_eio

```

Read an event from a generic `eio` event stream. The event stream element must specify the process within the sample (`i_prc`), the MC group for this process (`i_mci`), the selected term (`i_term`), the selected MC integration `channel`, and the particle set of the event.

We may encounter EOF, which we indicate by storing 0 for the process index `i_prc`. An I/O error will be reported, and we also abort reading.

*<Simulations: simulation: TBP>+≡*

```

generic :: read_event => read_event_eio
procedure :: read_event_eio => simulation_read_event_eio

```

*<Simulations: procedures>+≡*

```

subroutine simulation_read_event_eio (object, eio)
  class(simulation_t), intent(inout) :: object
  class(eio_t), intent(inout) :: eio
  integer :: iostat, current
  call eio%input_i_prc (current, iostat)
  select case (iostat)
  case (0)
    object%i_prc = current
    call eio%input_event (object%entry(current)%event_t, iostat)
  end select
  select case (iostat)
  case (:-1)
    object%i_prc = 0
    object%i_mci = 0
  case (1:)
    call msg_error ("Reading events: I/O error, aborting read")
    object%i_prc = 0
    object%i_mci = 0
  case default
    object%i_mci = object%entry(current)%get_i_mci ()
  end select
end subroutine simulation_read_event_eio

```

### 19.11.7 Event Stream Array

Write an event using an array of event I/O streams. The process index must be selected, or the current index must be available.

```
<Simulations: simulation: TBP>+≡
    generic :: write_event => write_event_es_array
    procedure :: write_event_es_array => simulation_write_event_es_array

<Simulations: procedures>+≡
    subroutine simulation_write_event_es_array (object, es_array)
        class(simulation_t), intent(in) :: object
        class(event_stream_array_t), intent(inout) :: es_array
        integer :: current
        current = object%i_prc
        if (current > 0) then
            call es_array%output (object%entry(current)%event_t, current)
        else
            call msg_fatal ("Simulation: write event: no process selected")
        end if
    end subroutine simulation_write_event_es_array
```

Read an event using an array of event I/O streams. Reading is successful if there is an input stream within the array, and if a valid event can be read from that stream. If there is a stream, but EOF is passed when reading the first item, we switch the channel to output and return failure but no error message, such that new events can be appended to that stream.

```
<Simulations: simulation: TBP>+≡
    generic :: read_event => read_event_es_array
    procedure :: read_event_es_array => simulation_read_event_es_array

<Simulations: procedures>+≡
    subroutine simulation_read_event_es_array (object, es_array, enable_switch, &
        fail)
        class(simulation_t), intent(inout) :: object
        class(event_stream_array_t), intent(inout) :: es_array
        logical, intent(in) :: enable_switch
        logical, intent(out) :: fail
        integer :: iostat, current
        if (es_array%has_input ()) then
            fail = .false.
            call es_array%input_i_prc (current, iostat)
            select case (iostat)
            case (0)
                object%i_prc = current
                call es_array%input_event (object%entry(current)%event_t, iostat)
            case (:-1)
                write (msg_buffer, "(A,1x,I0,1x,A)" ) &
                    "... event file terminates after", &
                    object%counter%read, "events."
                call msg_message ()
            if (enable_switch) then
                call es_array%switch_inout ()
                write (msg_buffer, "(A,1x,I0,1x,A)" ) &
                    "Generating remaining ", &
```



```

        object%n_evt_requested - object%counter%read, "events ..."
        call msg_message ()
    end if
    fail = .true.
    return
end select
select case (iostat)
case (0)
    object%i_mci = object%entry(current)%get_i_mci ()
case default
    write (msg_buffer, "(A,1x,I0,1x,A)") &
        "Reading events: I/O error, aborting read after", &
        object%counter%read, "events."
    call msg_error ()
    object%i_prc = 0
    object%i_mci = 0
    fail = .true.
end select
else
    fail = .true.
end if
end subroutine simulation_read_event_es_array

```

### 19.11.8 Recover event

Recalculate the process instance contents, given an event with known particle set. The indices for MC, term, and channel must be already set.

```

<Simulations: simulation: TBP>+≡
    procedure :: recalculate => simulation_recalculate

<Simulations: procedures>+≡
    subroutine simulation_recalculate (simulation, update_sqme)
        class(simulation_t), intent(inout) :: simulation
        logical, intent(in) :: update_sqme
        integer :: i_prc
        i_prc = simulation%i_prc
        call simulation%entry(i_prc)%recalculate (update_sqme)
        call simulation%entry(i_prc)%evaluate_expressions ()
    end subroutine simulation_recalculate

```

### 19.11.9 Auxiliary stuff

Write a separator line.

```

<Simulations: procedures>+≡
    subroutine write_separator (u)
        integer, intent(in) :: u
        write (u, "(A)") repeat ("-", 72)
    end subroutine write_separator

    subroutine write_separator_double (u)
        integer, intent(in) :: u

```

```

        write (u, "(A)") repeat ("=", 72)
    end subroutine write_separator_double

```

### 19.11.10 Extract contents

Return an array of pointers to the currently selected processes.

```

<Simulations: simulation: TBP>+≡
    procedure :: get_process_ptr => simulation_get_process_ptr

<Simulations: procedures>+≡
    function simulation_get_process_ptr (simulation) result (ptr)
        class(simulation_t), intent(in) :: simulation
        type(process_ptr_t), dimension(:), allocatable :: ptr
        integer :: i
        allocate (ptr (simulation%n_prc))
        do i = 1, size (ptr)
            ptr(i)%ptr => simulation%entry(i)%get_process_ptr ()
        end do
    end function simulation_get_process_ptr

```

Return the MD5 sum that summarizes configuration and integration (but not the event file). Used for initializing the event streams.

```

<Simulations: simulation: TBP>+≡
    procedure :: get_md5sum_prc => simulation_get_md5sum_prc
    procedure :: get_md5sum_cfg => simulation_get_md5sum_cfg

<Simulations: procedures>+≡
    function simulation_get_md5sum_prc (simulation) result (md5sum)
        class(simulation_t), intent(in) :: simulation
        character(32) :: md5sum
        md5sum = simulation%md5sum_prc
    end function simulation_get_md5sum_prc

    function simulation_get_md5sum_cfg (simulation) result (md5sum)
        class(simulation_t), intent(in) :: simulation
        character(32) :: md5sum
        md5sum = simulation%md5sum_cfg
    end function simulation_get_md5sum_cfg

```

Return data that may be useful for writing event files..

```

<Simulations: simulation: TBP>+≡
    procedure :: get_data => simulation_get_data

<Simulations: procedures>+≡
    function simulation_get_data (simulation) result (data)
        class(simulation_t), intent(in) :: simulation
        type(event_sample_data_t) :: data
        type(process_t), pointer :: process
        type(beam_data_t), pointer :: beam_data
        integer :: n, i
        process => simulation%entry(1)%get_process_ptr ()
        beam_data => process%get_beam_data_ptr ()
    end function simulation_get_data

```

```

call data%init (simulation%n_prc)
data%unweighted = simulation%unweighted
data%negative_weights = simulation%negative_weights
n = beam_data_get_n_in (beam_data)
data%n_beam = n
data%pdg_beam(:n) = flavor_get_pdg (beam_data_get_flavor (beam_data))
data%energy_beam(:n) = beam_data_get_energy (beam_data)
do i = 1, simulation%n_prc
  data%proc_num_id(i) = i
  data%cross_section(i) = simulation%entry(i)%integral
  data%error(i) = simulation%entry(i)%error
end do
data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
end function simulation_get_data

```

Return a default name for the current event sample. This is the process ID of the first process.

```

<Simulations: simulation: TBP>+≡
  procedure :: get_default_sample_name => simulation_get_default_sample_name
<Simulations: procedures>+≡
  function simulation_get_default_sample_name (simulation) result (sample)
    class(simulation_t), intent(in) :: simulation
    type(string_t) :: sample
    type(process_t), pointer :: process
    sample = "whizard"
    if (simulation%n_prc > 0) then
      process => simulation%entry(1)%get_process_ptr ()
      if (associated (process)) then
        sample = process%get_id ()
      end if
    end if
  end function simulation_get_default_sample_name

```

### 19.11.11 Test

This is the master for calling self-test procedures.

```

<Simulations: public>+≡
  public :: simulations_test
<Simulations: tests>≡
  subroutine simulations_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  <Simulations: execute tests>
  end subroutine simulations_test

```

### Initialization

Initialize a `simulation_t` object, including the embedded event records.

```

<Simulations: execute tests>≡

```

```

call test (simulations_1, "simulations_1", &
          "initialization", &
          u, results)
<Simulations: tests>+≡
subroutine simulations_1 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname1, procname2
  type(rt_data_t), target :: global
  type(simulation_t), target :: simulation

  write (u, "(A)")  "* Test output: simulations_1"
  write (u, "(A)")  "* Purpose: initialize simulation"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize processes"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)

  libname = "simulation_1a"
  procname1 = "simulation_1p"

  call prepare_test_library (global, libname, 1, [procname1])
  call compile_library (libname, global)

  call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$phs_method"), &
    var_str ("single"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("integration_method"), &
    var_str ("midpoint"), is_known = .true.)

  call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

  call global%it_list%init ([1], [1000])

  call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("simulations1"), is_known = .true.)
  call integrate_process (procname1, global)

  libname = "simulation_1b"
  procname2 = "sim_extra"

  call prepare_test_library (global, libname, 1, [procname2])
  call compile_library (libname, global)
  call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("simulations2"), is_known = .true.)

```

```

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("sample"), &
    var_str ("sim1"), is_known = .true.)
call integrate_process (procname2, global)

call simulation%init ([procname1, procname2], global)
call simulation%compute_weights ()
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the first process"
write (u, "(A)")

call simulation%write_event (u, i_prc = 1)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_1"

end subroutine simulations_1

```

## Weighted events

Generate events for a single process.

```

<Simulations: execute tests>+≡
    call test (simulations_2, "simulations_2", &
        "weighted events", &
        u, results)

<Simulations: tests>+≡
subroutine simulations_2 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1
    type(rt_data_t), target :: global
    type(simulation_t), target :: simulation
    type(event_sample_data_t) :: data

    write (u, "(A)")  "* Test output: simulations_2"
    write (u, "(A)")  "* Purpose: generate events for a single process"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize processes"
    write (u, "(A)")

    call syntax_model_file_init ()

```

```

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
    .false., is_known = .true.)

libname = "simulation_2a"
procname1 = "simulation_2p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("phs_method"), &
    var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("integration_method"), &
    var_str ("midpoint"), is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("run_id"), &
    var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data = simulation%get_data ()
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate three events"
write (u, "(A)")

call simulation%generate (3)
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the last event"
write (u, "(A)")

call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

```

```

call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_2"

end subroutine simulations_2

```

## Unweighted events

Generate events for a single process.

```

<Simulations: execute tests>+=
  call test (simulations_3, "simulations_3", &
    "unweighted events", &
    u, results)

<Simulations: tests>+=
  subroutine simulations_3 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1
    type(rt_data_t), target :: global
    type(simulation_t), target :: simulation
    type(event_sample_data_t) :: data

    write (u, "(A)")  "* Test output: simulations_3"
    write (u, "(A)")  "* Purpose: generate unweighted events &
      &for a single process"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize processes"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
      .false., is_known = .true.)

    libname = "simulation_3a"
    procname1 = "simulation_3p"

    call prepare_test_library (global, libname, 1, [procname1])
    call compile_library (libname, global)

    call var_list_set_string (global%var_list, var_str ("method"), &
      var_str ("unit_test"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("phs_method"), &
      var_str ("single"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("integration_method"), &
      var_str ("midpoint"), is_known = .true.)

    call var_list_set_real (global%var_list, var_str ("sqrts"), &

```

```

1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call simulation%init ([procname1], global)
call simulation%compute_weights ()

data = simulation%get_data ()
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate three events"
write (u, "(A)")

call simulation%generate (3)
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the last event"
write (u, "(A)")

call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_3"

end subroutine simulations_3

```

## Simulating process with structure functions

Generate events for a single process.

```

<Simulations: execute tests>+≡
    call test (simulations_4, "simulations_4", &
        "process with structure functions", &
        u, results)

<Simulations: tests>+≡
    subroutine simulations_4 (u)
        integer, intent(in) :: u

```



```

type(string_t) :: libname, procname1
type(rt_data_t), target :: global
type(flavor_t) :: flv
type(simulation_t), target :: simulation
type(event_sample_data_t) :: data

write (u, "(A)")  "* Test output: simulations_4"
write (u, "(A)")  "* Purpose: generate events for a single process &
                  &with structure functions"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
                      .false., is_known = .true.)

libname = "simulation_4a"
procname1 = "simulation_4p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
                          var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_method"), &
                          var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_phs_method"), &
                          var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_integration_method"), &
                          var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
                      .true., is_known = .true.)
call var_list_append_log (global%var_list, var_str ("?rebuild_grids"), &
                          .true., intrinsic = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
                       1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
                       0._default, is_known = .true.)

call reset_interaction_counter ()

call flavor_init (flv, 25, global%model)
call beam_data_init_sqrts (global%beam_data, &
                          var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

write (u, "(A)")  "* Integrate"

```

```

write (u, "(A)")

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
    var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$_sample"), &
    var_str ("simulations4"), is_known = .true.)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data = simulation%get_data ()
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate three events"
write (u, "(A)")

call simulation%generate (3)
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the last event"
write (u, "(A)")

call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_4"

end subroutine simulations_4

```

## Event I/O

Generate event for a test process, write to file and reread.

```

<Simulations: execute tests>+≡
call test (simulations_5, "simulations_5", &
    "raw event I/O", &
    u, results)

```

*<Simulations: tests>+≡*

```

subroutine simulations_5 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname1, sample
  type(rt_data_t), target :: global
  type(process_ptr_t) :: process_ptr
  class(eio_t), allocatable :: eio
  type(simulation_t), allocatable, target :: simulation

  write (u, "(A)")  "* Test output: simulations_5"
  write (u, "(A)")  "*   Purpose: generate events for a single process"
  write (u, "(A)")  "*               write to file and reread"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize processes"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)

  libname = "simulation_5a"
  procname1 = "simulation_5p"

  call prepare_test_library (global, libname, 1, [procname1])
  call compile_library (libname, global)

  call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$phs_method"), &
    var_str ("single"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$integration_method"), &
    var_str ("midpoint"), is_known = .true.)

  call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)

  call global%it_list%init ([1], [1000])

  call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("simulations5"), is_known = .true.)
  call integrate_process (procname1, global)

  write (u, "(A)")  "* Initialize event generation"
  write (u, "(A)")

  call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
  sample = "simulations5"
  call var_list_set_string (global%var_list, var_str ("$sample"), &
    sample, is_known = .true.)
  allocate (simulation)

```

```

call simulation%init ([procname1], global)
call simulation%compute_weights ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

process_ptr%ptr => global%process_stack%get_process_ptr (procname1)

allocate (eio_raw_t :: eio)
call eio%init_out (sample, [process_ptr])

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call simulation%write_event (u)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample, [process_ptr])

call simulation%read_event (eio)
call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Recalculate process instance"
write (u, "(A)")

call simulation%recalculate (update_sqme = .true.)
call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_5"

```

```
end subroutine simulations_5
```

## Event I/O

Generate event for a real process with structure functions, write to file and reread.

```
<Simulations: execute tests>+≡
  call test (simulations_6, "simulations_6", &
    "raw event I/O with structure functions", &
    u, results)

<Simulations: tests>+≡
  subroutine simulations_6 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    type(process_ptr_t) :: process_ptr
    class(eio_t), allocatable :: eio
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv

    write (u, "(A)")  "* Test output: simulations_6"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*               write to file and reread"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and integrate"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
      .false., is_known = .true.)

    libname = "simulation_6"
    procname1 = "simulation_6p"

    call prepare_test_library (global, libname, 1, [procname1])
    call compile_library (libname, global)

    call var_list_set_string (global%var_list, var_str ("method"), &
      var_str ("unit_test"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("phs_method"), &
      var_str ("wood"), is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("integration_method"), &
      var_str ("vamp"), is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
      .true., is_known = .true.)
    call var_list_append_log (global%var_list, var_str ("?rebuild_grids"), &
      .true., intrinsic = .true.)

    call var_list_set_real (global%var_list, var_str ("qrts"), &
```

```

1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call beam_data_init_sqrts (global%beam_data, &
var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$_run_id"), &
var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
.false., is_known = .true.)
sample = "simulations6"
call var_list_set_string (global%var_list, var_str ("$_sample"), &
sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

process_ptr%ptr => global%process_stack%get_process_ptr (procname1)

allocate (eio_raw_t :: eio)
call eio%init_out (sample, [process_ptr])

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

```

```

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample, [process_ptr])

call simulation%read_event (eio)
call simulation%write_event (u, verbose = .true.)

write (u, "(A)")
write (u, "(A)")  "* Recalculate process instance"
write (u, "(A)")

call simulation%recalculate (update_sqme = .true.)
call simulation%write_event (u, verbose = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_6"

end subroutine simulations_6

```

## Automatic Event I/O

Generate events with raw-format event file as cache: generate, reread, append.

```

<Simulations: execute tests>+≡
call test (simulations_7, "simulations_7", &
  "automatic raw event I/O", &
  u, results)

<Simulations: tests>+≡
subroutine simulations_7 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname1, sample
  type(rt_data_t), target :: global
  type(string_t), dimension(0) :: empty_string_array
  type(event_sample_data_t) :: data
  type(event_stream_array_t) :: es_array
  type(simulation_t), allocatable, target :: simulation
  type(flavor_t) :: flv

  write (u, "(A)")  "* Test output: simulations_7"
  write (u, "(A)")  "* Purpose: generate events for a single process"

```

```

write (u, "(A)")  "*"           write to file and reread"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and integrate"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
    .false., is_known = .true.)

libname = "simulation_7"
procname1 = "simulation_7p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
    var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_append_log (global%var_list, var_str ("?rebuild_grids"), &
    .true., intrinsic = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .true., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
    0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call beam_data_init_sqrts (global%beam_data, &
    var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
sample = "simulations7"

```



```

call var_list_set_string (global%var_list, var_str ("sample"), &
    sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init &
    (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
    data)

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Re-read the event from file and generate another one"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, &
    empty_string_array, simulation%get_process_ptr (), global, data, &
    input = var_str ("raw"))

call simulation%generate (2, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read both events from file"
write (u, "(A)")

call reset_interaction_counter ()

```

```

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, &
    empty_string_array, simulation%get_process_ptr (), global, data, &
    input = var_str ("raw"))

call simulation%generate (2, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call es_array%final ()
call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_7"

end subroutine simulations_7

```

## Rescanning Events

Generate events and rescan the resulting raw event file.

```

<Simulations: execute tests>+≡
    call test (simulations_8, "simulations_8", &
        "rescan raw event file", &
        u, results)

<Simulations: tests>+≡
subroutine simulations_8 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    type(string_t), dimension(0) :: empty_string_array
    type(event_sample_data_t) :: data
    type(event_stream_array_t) :: es_array
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv

    write (u, "(A)")  "* Test output: simulations_8"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*               write to file and rescan"
    write (u, "(A)")

```

```

write (u, "(A)")  "* Initialize process and integrate"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
    .false., is_known = .true.)

libname = "simulation_8"
procname1 = "simulation_8p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("method"), &
    var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("phs_method"), &
    var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("integration_method"), &
    var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
    .true., is_known = .true.)
call var_list_append_log (global%var_list, var_str ("?rebuild_grids"), &
    .true., intrinsic = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
    0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call beam_data_init_sqrts (global%beam_data, &
    var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("run_id"), &
    var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
sample = "simulations8"
call var_list_set_string (global%var_list, var_str ("sample"), &
    sample, is_known = .true.)
allocate (simulation)

```

```

call simulation%init ([procname1], global)
call simulation%compute_weights ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc)  = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init &
    (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
    data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)  = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, &
    empty_string_array, simulation%get_process_ptr (), global, data, &
    input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

call simulation%rescan (es_array, &
    update_event = .false., &
    update_sqme = .false., &
    update_weight = .false., &
    model = global%model, &
    helicity_selection = global%get_helicity_selection ())

write (u, "(A)")

```

```

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read again and recalculate"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)  = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, &
    empty_string_array, simulation%get_process_ptr (), global, data, &
    input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

call simulation%rescan (es_array, &
    update_event = .true., &
    update_sqme = .true., &
    update_weight = .false., &
    model = global%model, &
    helicity_selection = global%get_helicity_selection ())

write (u, "(A)")

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call es_array%final ()
call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_8"

end subroutine simulations_8

```

## Rescanning Check

Generate events and rescan with process mismatch.

```

<Simulations: execute tests>+≡
    call test (simulations_9, "simulations_9", &
        "rescan mismatch", &
        u, results)

<Simulations: tests>+≡
    subroutine simulations_9 (u)
        integer, intent(in) :: u
        type(string_t) :: libname, procname1, sample
        type(rt_data_t), target :: global
        type(string_t), dimension(0) :: empty_string_array
        type(event_sample_data_t) :: data
        type(event_stream_array_t) :: es_array
        type(simulation_t), allocatable, target :: simulation
        type(flavor_t) :: flv
        logical :: error

        write (u, "(A)")  "* Test output: simulations_9"
        write (u, "(A)")  "*   Purpose: generate events for a single process"
        write (u, "(A)")  "*               write to file and rescan"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize process and integrate"
        write (u, "(A)")

        call syntax_model_file_init ()

        call global%global_init ()
        call var_list_set_log (global%var_list, var_str ("?omega_omp"), &
            .false., is_known = .true.)

        libname = "simulation_9"
        procname1 = "simulation_9p"

        call prepare_test_library (global, libname, 1, [procname1])
        call compile_library (libname, global)

        call var_list_set_string (global%var_list, var_str ("$method"), &
            var_str ("unit_test"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$phs_method"), &
            var_str ("wood"), is_known = .true.)
        call var_list_set_string (global%var_list, var_str ("$integration_method"), &
            var_str ("vamp"), is_known = .true.)
        call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"), &
            .true., is_known = .true.)
        call var_list_append_log (global%var_list, var_str ("?rebuild_grids"), &
            .true., intrinsic = .true.)

        call var_list_set_real (global%var_list, var_str ("sqrts"), &
            1000._default, is_known = .true.)
        call var_list_set_real (global%var_list, var_str ("ms"), &
            0._default, is_known = .true.)

        call flavor_init (flv, 25, global%model)
        call beam_data_init_sqrts (global%beam_data, &

```

```

        var_list_get_rval (global%var_list, var_str ("sqrts")), [flv, flv])
call global%beam_structure%init ([1], global%var_list)
call global%beam_structure%set_entry (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
    var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
    .false., is_known = .true.)
sample = "simulations9"
call var_list_set_string (global%var_list, var_str ("$sample"), &
    sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], global)
call simulation%compute_weights ()

call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc) = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init &
    (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
    data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Initialize event generation for different parameters"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)

```

```

call simulation%init ([procname1, procname1], global)
call simulation%compute_weights ()

call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Attempt to re-read the events (should fail)"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, &
    empty_string_array, simulation%get_process_ptr (), global, data, &
    input = var_str ("raw"), input_sample = sample, &
    allow_switch = .false., error = error)

write (u, "(1x,A,L1)")  "error = ", error

call simulation%rescan (es_array, &
    update_event = .false., &
    update_sqme = .false., &
    update_weight = .false., &
    model = global%model, &
    helicity_selection = global%get_helicity_selection ())

call es_array%final ()
call simulation%final ()
call global%final ()
call model_list_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_9"

end subroutine simulations_9

```



# Chapter 20

## Top level API

### 20.1 Commands

This module defines the command language of the main input file.

`<commands.f90>`≡  
*<File header>*

```
module commands

  <Use kinds>
  <Use strings>
  use constants !NODEP!
  <Use file utils>
  ! use limits, only: HISTOGRAM_DATA_FORMAT !NODEP!
  ! use limits, only: DEFAULT_ANALYSIS_FILENAME !NODEP!
  ! use limits, only: FORBIDDEN_ENDINGS1 !NODEP!
  ! use limits, only: FORBIDDEN_ENDINGS2 !NODEP!
  ! use limits, only: FORBIDDEN_ENDINGS3 !NODEP!
  use diagnostics !NODEP!
  use unit_tests
  ! use lorentz !NODEP!
  use sm_qcd
  ! use tao_random_numbers !NODEP!
  use pdf_builtin !NODEP!
  use sf_lhapdf
  ! use md5
  ! use os_interface
  use ifiles
  use lexers
  use syntax_rules
  use parser
  ! use analysis
  use pdg_arrays
  ! use subevents
  use variables
  use expressions
  use models
  ! use state_matrices
```

```

    use interactions
    use flavors
    !   use quantum_numbers
    !   use polarizations
    use beams
    !   use strfun
    !   use mappings
    !   use phs_forests
    !   use cascades
    use particle_specifiers
    use process_libraries
    use prclib_stacks
    !   use processes
    !   use decays
    !   use events
    !   use slha_interface
    !   use cputime
    !   use iterations
    !   use beam_polarizations
    !   use strfun_config
    !   use user_files
    use eio_data
    use rt_data
    use dispatch
    use process_configurations
    use compilations
    use integrations
    use event_streams
    use simulations
    !   use nlo_setup
    !   use core_interactions_config

```

*⟨Standard module head⟩*

*⟨Commands: public⟩*

*⟨Commands: types⟩*

*⟨Commands: variables⟩*

*⟨Commands: interfaces⟩*

**contains**

*⟨Commands: procedures⟩*

*⟨Commands: tests⟩*

**end module commands**

### 20.1.1 The command type

The command type is a generic type that holds any command, compiled for execution.

```

<Commands: types>≡
  type, abstract :: command_t
    type(parse_node_t), pointer :: pn => null ()
    class(command_t), pointer :: next => null ()
  contains
    <Commands: command: TBP>
  end type command_t

```

Finalizer: Default is to do nothing.

```

<Commands: command: TBP>≡
  procedure :: final => command_final

<Commands: procedures>≡
  subroutine command_final (command)
    class(command_t), intent(inout) :: command
  end subroutine command_final

```

Allocate a command with the appropriate concrete type. Store the parse node pointer in the command object, so we can reference to it when compiling.

```

<Commands: procedures>+≡
  subroutine dispatch_command (command, pn)
    class(command_t), intent(inout), pointer :: command
    type(parse_node_t), intent(in), target :: pn
    select case (char (parse_node_get_rule_key (pn)))
    case ("cmd_model")
      allocate (cmd_model_t :: command)
    case ("cmd_library")
      allocate (cmd_library_t :: command)
    case ("cmd_process")
      allocate (cmd_process_t :: command)
    case ("cmd_compile")
      allocate (cmd_compile_t :: command)
    ! case ("cmd_load")
    !   command%type = CMD_LOAD
    !   call cmd_load_compile (command%load, pn, global)
    ! case ("cmd_exec")
    !   command%type = CMD_EXEC
    !   call cmd_exec_compile (command%exec, pn, global)
    case ("cmd_num", "cmd_complex", "cmd_real", "cmd_int", &
      "cmd_log_decl", "cmd_log", "cmd_string", "cmd_string_decl", &
      "cmd_alias", "cmd_result")
      allocate (cmd_var_t :: command)
    ! case ("cmd_slha")
    !   command%type = CMD_SLHA
    !   call cmd_slha_compile (command%slha, pn, global)
    ! case ("cmd_show")
    !   command%type = CMD_SHOW
    !   call cmd_show_compile (command%show, pn, global)
    ! case ("cmd_expect")
    !   command%type = CMD_EXPECT
    !   call cmd_expect_compile (command%expect, pn, global)
    case ("cmd_beams")
      allocate (cmd_beams_t :: command)

```

```

!     case ("cmd_beam_polarization")
!         command%type = CMD_BEAM_POLARIZATION
!         call cmd_beam_polarization_compile &
!             (command%beam_polarization, pn, global)
case ("cmd_cuts")
    allocate (cmd_cuts_t :: command)
!     case ("cmd_scale")
!         command%type = CMD_SCALE
!         call cmd_scale_compile (command%scale, pn)
!     case ("cmd_fac_scale")
!         command%type = CMD_FAC_SCALE
!         call cmd_fac_scale_compile (command%fac_scale, pn)
!     case ("cmd_ren_scale")
!         command%type = CMD_REN_SCALE
!         call cmd_ren_scale_compile (command%ren_scale, pn)
!     case ("cmd_weight")
!         command%type = CMD_WEIGHT
!         call cmd_weight_compile (command%weight, pn)
!     case ("cmd_selection")
!         command%type = CMD_SELECTION
!         call cmd_selection_compile (command%selection, pn)
!     case ("cmd_reweight")
!         command%type = CMD_REWEIGHT
!         call cmd_reweight_compile (command%reweight, pn)
!     case ("cmd_seed")
!         command%type = CMD_SEED
!         call cmd_seed_compile (command%seed, pn, global)
case ("cmd_iterations")
    allocate (cmd_iterations_t :: command)
case ("cmd_integrate")
    allocate (cmd_integrate_t :: command)
!     case ("cmd_me_test")
!         command%type = CMD_ME_TEST
!         call cmd_me_test_compile (command%me_test, pn, global)
!     case ("cmd_observable")
!         command%type = CMD_OBSERVABLE
!         call cmd_observable_compile (command%observable, pn, global)
!     case ("cmd_histogram")
!         command%type = CMD_HISTOGRAM
!         call cmd_histogram_compile (command%histogram, pn, global)
!     case ("cmd_plot")
!         command%type = CMD_PLOT
!         call cmd_plot_compile (command%plot, pn, global)
!     case ("cmd_graph")
!         command%type = CMD_GRAPH
!         call cmd_graph_compile (command%graph, pn, global)
!     case ("cmd_clear")
!         command%type = CMD_CLEAR
!         call cmd_clear_compile (command%clear, pn, global)
!     case ("cmd_record")
!         command%type = CMD_RECORD
!         call cmd_record_compile (command%record, pn, global)
!     case ("cmd_analysis")
!         command%type = CMD_ANALYSIS

```

```

!       call cmd_analysis_compile (command%analysis, pn)
!   case ("cmd_unstable")
!       command%type = CMD_UNSTABLE
!       call cmd_unstable_compile (command%unstable, pn, global)
!   case ("cmd_stable")
!       command%type = CMD_STABLE
!       call cmd_stable_compile (command%stable, pn, global)
!   case ("cmd_polarized")
!       command%type = CMD_POLARIZED
!       call cmd_un_polarized_compile (command%un_polarized, pn, global)
!   case ("cmd_unpolarized")
!       command%type = CMD_UNPOLARIZED
!       call cmd_un_polarized_compile (command%un_polarized, pn, global)
!   case ("cmd_sample_format")
!       allocate (cmd_sample_format_t :: command)
!   case ("cmd_simulate")
!       allocate (cmd_simulate_t :: command)
!   case ("cmd_rescan")
!       allocate (cmd_rescan_t :: command)
!   case ("cmd_write_analysis")
!       command%type = CMD_WRITE_ANALYSIS
!       call cmd_write_analysis_compile (command%write_analysis, &
!           pn, global)
!   case ("cmd_compile_analysis")
!       command%type = CMD_COMPILE_ANALYSIS
!       call cmd_write_analysis_compile (command%compile_analysis, pn, global)
!   case ("cmd_histogram_writer", "cmd_plot_writer")
!       command%type = CMD_HISTOGRAM_WRITER
!       call cmd_xxx_writer_compile (command%xxx_writer, pn, global)
!   case ("cmd_open_out")
!       command%type = CMD_OPEN_OUT
!       call cmd_open_out_compile (command%open_out, pn, global)
!   case ("cmd_close_out")
!       command%type = CMD_CLOSE_OUT
!       call cmd_close_out_compile (command%close_out, pn, global)
!   case ("cmd_printd")
!       command%type = CMD_PRINTD
!       call cmd_printd_compile (command%printd, pn, global)
!   case ("cmd_printf")
!       command%type = CMD_PRINTF
!       call cmd_printf_compile (command%printf, pn, global)
!   case ("cmd_scan")
!       command%type = CMD_SCAN
!       call cmd_scan_compile (command%loop, pn, global)
!   case ("cmd_if")
!       command%type = CMD_IF
!       call cmd_if_compile (command%cond, pn, global)
!   case ("cmd_include")
!       command%type = CMD_INCLUDE
!       call cmd_include_compile (command%include, pn, global)
!   case ("cmd_quit")
!       command%type = CMD_QUIT
!       call cmd_quit_compile (command%quit, pn, global)
!   case ("cmd_set_alpha_qed")

```

```

!      command%type = CMD_SET_ALPHA_QED
!      call cmd_set_alpha_qed_compile (command%set_alpha_qed, pn, global)
!      case ("cmd_soft_mass_regulator", "cmd_collinear_mass_regulators", &
!          "cmd_mask", "cmd_charges", "cmd_clear_mask", &
!          "cmd_clear_collinear_mass_regulators", "cmd_clear_charges")
!          command%type = CMD_CHANGE_DIPOLE_CONFIG
!          call cmd_change_nlo_setup_compile (command%change_nlo_setup, &
!              pn, global)
!      case ("cmd_nlo_setup")
!          command%type = CMD_NLO_SETUP
!          call cmd_nlo_setup_compile (command%nlo_setup, pn, global)
!      case ("cmd_process_sum")
!          command%type = CMD_PROCESS_SUM
!          call cmd_process_sum_compile (command%process_sum, pn, global)
!      case default
!          print *, char (parse_node_get_rule_key (pn))
!          call msg_bug ("Command not implemented")
!      end select
!      command%pn => pn
!  end subroutine dispatch_command

```

Output. We allow for indentation so we can display a command tree.

```

⟨Commands: command: TBP⟩+=
  procedure (command_write), deferred :: write

⟨Commands: interfaces⟩=
  abstract interface
    subroutine command_write (cmd, unit, indent)
      import
      class(command_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
    end subroutine command_write
  end interface

```

Compile a command. The command type is already fixed, so this is a deferred type-bound procedure.

```

⟨Commands: command: TBP⟩+=
  procedure (command_compile), deferred :: compile

⟨Commands: interfaces⟩+=
  abstract interface
    subroutine command_compile (cmd, global)
      import
      class(command_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
    end subroutine command_compile
  end interface

```

Execute a command. This will use and/or modify the runtime data set. If the quit flag is set, the caller should terminate command execution.

```

⟨Commands: command: TBP⟩+=
  procedure (command_execute), deferred :: execute

```

```

⟨Commands: interfaces⟩+=
  abstract interface
    subroutine command_execute (cmd, global)
      import
      class(command_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
    end subroutine command_execute
  end interface

```

Auxiliary for output:

```

⟨Commands: procedures⟩+=
  subroutine write_indent (unit, indent)
    integer, intent(in) :: unit
    integer, intent(in), optional :: indent
    if (present (indent)) then
      write (unit, "(A)", advance="no") repeat (" ", indent)
    end if
  end subroutine write_indent

```

## 20.1.2 Specific command types

### Model configuration

The command declares a model, looks for the specified file and loads it.

```

⟨Commands: types⟩+=
  type, extends (command_t) :: cmd_model_t
  private
    type(string_t) :: name
  contains
    ⟨Commands: cmd model: TBP⟩
  end type cmd_model_t

```

Output

```

⟨Commands: cmd model: TBP⟩=
  procedure :: write => cmd_model_write

⟨Commands: procedures⟩+=
  subroutine cmd_model_write (cmd, unit, indent)
    class(cmd_model_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    write (u, "(1x,A,1x,'"',A,'\"') "model =", char (cmd%name)
  end subroutine cmd_model_write

```

Compile. Get the model name and read the model from file, so it is readily available when the command list is executed.

```

⟨Commands: cmd model: TBP⟩+=
  procedure :: compile => cmd_model_compile

```

*<Commands: procedures>+≡*

```

subroutine cmd_model_compile (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_name
  type(model_t), pointer :: mdl
  type(string_t) :: filename
  pn_name => parse_node_get_sub_ptr (cmd%pn, 3)
  cmd%name = parse_node_get_string (pn_name)
  filename = cmd%name // ".mdl"
  mdl => null ()
  call model_list_read_model (cmd%name, filename, global%os_data, mdl)
  if (associated (mdl)) then
    call var_list_init_copies &
      (global%var_list, model_get_var_list_ptr (mdl))
  end if
end subroutine cmd_model_compile

```

Execute: Insert a pointer into the global data record and reassign the variable list.

*<Commands: cmd model: TBP>+≡*

```

procedure :: execute => cmd_model_execute

```

*<Commands: procedures>+≡*

```

subroutine cmd_model_execute (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(model_t), pointer :: mdl
  type(var_list_t), pointer :: model_vars
  logical :: new_model
  new_model = .true.
  if (associated (global%model)) then
    new_model = model_get_name (global%model) /= cmd%name
  end if
  if (new_model) then
    if (model_list_model_exists (cmd%name)) then
      mdl => model_list_get_model_ptr (cmd%name)
      global%model => mdl
      call var_list_set_string (global%var_list, var_str ("$model_name"), &
        cmd%name, is_known=.true.)
      call msg_message ("Switching to model '" &
        // char (model_get_name (mdl)) &
        // "': reassigning model parameters")
      model_vars => model_get_var_list_ptr (mdl)
      call var_list_synchronize &
        (global%var_list, model_vars, reset_pointers = .true.)
    end if
  else
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_synchronize &
      (global%var_list, model_vars, reset_pointers = .false.)
  end if
end subroutine cmd_model_execute

```



## Library configuration

We configure a process library that should hold the subsequently defined processes. If the referenced library exists already, just make it the currently active one.

```
<Commands: types>+≡
    type, extends (command_t) :: cmd_library_t
        private
        type(string_t) :: name
    contains
        <Commands: cmd library: TBP>
    end type cmd_library_t
```

Output.

```
<Commands: cmd library: TBP>≡
    procedure :: write => cmd_library_write

<Commands: procedures>+≡
    subroutine cmd_library_write (cmd, unit, indent)
        class(cmd_library_t), intent(in) :: cmd
        integer, intent(in), optional :: unit, indent
        integer :: u
        u = output_unit (unit)
        call write_indent (u, indent)
        write (u, "(1x,A,1x,'"',A,'\"')") "library =", char (cmd%name)
    end subroutine cmd_library_write
```

Compile. Get the library name.

```
<Commands: cmd library: TBP>+≡
    procedure :: compile => cmd_library_compile

<Commands: procedures>+≡
    subroutine cmd_library_compile (cmd, global)
        class(cmd_library_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(parse_node_t), pointer :: pn_name
        pn_name => parse_node_get_sub_ptr (cmd%pn, 3)
        cmd%name = parse_node_get_string (pn_name)
    end subroutine cmd_library_compile
```

Execute: Initialize a new library and push it on the library stack (if it does not yet exist). Insert a pointer to the library into the global data record. Then, try to load the library unless the `rebuild` flag is set.

```
<Commands: cmd library: TBP>+≡
    procedure :: execute => cmd_library_execute

<Commands: procedures>+≡
    subroutine cmd_library_execute (cmd, global)
        class(cmd_library_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(prclib_entry_t), pointer :: lib_entry
        type(process_library_t), pointer :: lib
        logical :: rebuild_library, recompile_library
```

```

lib => global%prclib_stack%get_library_ptr (cmd%name)
!!! JRR: WK please check
rebuild_library = &
    var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
!!! recompile_library = &
!!!    var_list_get_lval (global%var_list, var_str ("?recompile_library"))
if (.not. (associated (lib))) then
    allocate (lib_entry)
    call lib_entry%init (cmd%name)
    lib => lib_entry%process_library_t
    call global%add_prclib (lib_entry)
else
    call global%update_prclib (lib)
end if
if (associated (lib) .and. .not. rebuild_library) then
    call lib%update_status (global%os_data)
end if
end subroutine cmd_library_execute

```

## Dipole setup

```

<CCC Commands: types>≡
type :: cmd_set_alpha_qed_t
    private
    type(parse_node_t), pointer :: alpha_qed_pn => null ()
end type cmd_set_alpha_qed_t

<CCC Commands: procedures>≡
subroutine cmd_set_alpha_qed_compile (cfg, pn, global)
    type(cmd_set_alpha_qed_t), pointer :: cfg
    type(parse_node_t), intent(in) :: pn
    type(rt_data_t), intent(inout), target :: global
    allocate (cfg)
    cfg%alpha_qed_pn => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_set_alpha_qed_compile

<CCC Commands: procedures>+≡
subroutine cmd_set_alpha_qed_execute (cfg, global)
    type(cmd_set_alpha_qed_t), intent(inout), target :: cfg
    type(rt_data_t), intent(inout), target :: global
    global%pn_alpha_qed_expr => cfg%alpha_qed_pn
end subroutine cmd_set_alpha_qed_execute

<CCC Commands: types>+≡
type cmd_change_nlo_setup_t
    private
    integer :: type
    type(parse_node_t), pointer :: pn_mreg => null ()
    type(parse_node_p), dimension(:), allocatable :: pn_masses
    type(parse_node_p), dimension(:), allocatable :: pn_mask
    type(parse_node_p), dimension(:), allocatable :: pn_charges
end type cmd_change_nlo_setup_t

```

{CCC Commands: procedures}+≡

```

subroutine cmd_change_nlo_setup_compile (change_config, pn, global)
  type(cmd_change_nlo_setup_t), pointer :: change_config
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_vals
  integer :: nvals, i
  allocate (change_config)
  pn_vals => parse_node_get_sub_ptr (pn, 3)
  if (associated (pn_vals)) nvals = parse_node_get_n_sub (pn_vals)
  select case (char (parse_node_get_rule_key (pn)))
    case ("cmd_soft_mass_regulator")
      change_config%type = NLO_SETUP_SET_MREG
      change_config%pn_mreg => pn_vals
    case ("cmd_collinear_mass_regulators")
      change_config%type = NLO_SETUP_SET_MASSES
      allocate (change_config%pn_masses(nvals))
      do i = 1, nvals
        change_config%pn_masses(i)%ptr => parse_node_get_sub_ptr (pn_vals, i)
      end do
    case ("cmd_mask")
      change_config%type = NLO_SETUP_SET_MASK
      allocate (change_config%pn_mask(nvals))
      do i = 1, nvals
        change_config%pn_mask(i)%ptr => parse_node_get_sub_ptr (pn_vals, i)
      end do
    case ("cmd_charges")
      change_config%type = NLO_SETUP_SET_CHARGES
      allocate (change_config%pn_charges (nvals))
      do i = 1, nvals
        change_config%pn_charges(i)%ptr => parse_node_get_sub_ptr (pn_vals, i)
      end do
    case ("cmd_clear_collinear_mass_regulators")
      change_config%type = NLO_SETUP_CLEAR_MASSES
    case ("cmd_clear_mask")
      change_config%type = NLO_SETUP_CLEAR_MASK
    case ("cmd_clear_charges")
      change_config%type = NLO_SETUP_CLEAR_CHARGES
    case default
      call msg_bug ("unknown dipole setup command. This is a BUG.")
  end select
end subroutine cmd_change_nlo_setup_compile

```

{CCC Commands: procedures}+≡

```

subroutine cmd_change_nlo_setup_execute (change_config, global)
  type(cmd_change_nlo_setup_t), pointer :: change_config
  type(rt_data_t), intent(inout), target :: global
  real(kind=default) :: val
  real(kind=default), dimension(:), allocatable :: vals_real
  integer, dimension(:), allocatable :: vals_int
  type(parse_node_t), pointer :: fst, snd
  integer :: i
  select case (change_config%type)
    case (NLO_SETUP_SET_MREG)
      val = eval_real (change_config%pn_mreg, global%var_list)

```

```

        call nlo_setup_list_append (global%nlo_setup_list, &
            change_config%type, mreg=val)
    case (NLO_SETUP_SET_MASSES)
        allocate (vals_real (size (change_config%pn_masses)))
        do i = 1, size (vals_real)
            vals_real(i) = eval_real (change_config%pn_masses(i)%ptr, &
                global%var_list)
        end do
        call nlo_setup_list_append (global%nlo_setup_list, &
            change_config%type, masses=vals_real)
    case (NLO_SETUP_SET_MASK)
        allocate (vals_int(2*size (change_config%pn_mask)))
        do i = 1, size (change_config%pn_mask)
            fst => parse_node_get_sub_ptr (change_config%pn_mask(i)%ptr)
            snd => parse_node_get_next_ptr (fst, 2)
            vals_int(2*(i-1)+1) = eval_real (fst, global%var_list)
            vals_int(2*(i-1)+2) = eval_real (snd, global%var_list)
        end do
        call nlo_setup_list_append (global%nlo_setup_list, &
            change_config%type, mask=vals_int)
    case (NLO_SETUP_SET_CHARGES)
        allocate (vals_real (size (change_config%pn_charges)))
        do i = 1, size (change_config%pn_charges)
            vals_real(i) = eval_real (change_config%pn_charges(i)%ptr, &
                global%var_list)
        end do
        call nlo_setup_list_append (global%nlo_setup_list, &
            change_config%type, charges=vals_real)
    case (NLO_SETUP_CLEAR_MASSES, NLO_SETUP_CLEAR_MASK, &
        NLO_SETUP_CLEAR_CHARGES)
        call nlo_setup_list_append (global%nlo_setup_list, &
            change_config%type)
    case default
        call msg_bug ("invalid NLO setup command type. This is a BUG.")
    end select
end subroutine cmd_change_nlo_setup_execute

<CCC Commands: types>+≡
type cmd_nlo_setup_t
private
type(string_t) :: id
type(command_list_t), pointer :: options => null ()
type(rt_data_t) :: local
end type cmd_nlo_setup_t

<CCC Commands: procedures>+≡
subroutine cmd_nlo_setup_final (nlo_setup)
type(cmd_nlo_setup_t), intent(inout) :: nlo_setup
if (associated (nlo_setup%options)) then
    call command_list_final (nlo_setup%options)
    deallocate (nlo_setup%options)
end if
end subroutine cmd_nlo_setup_final

```

{CCC Commands: procedures}+≡

```

subroutine cmd_nlo_setup_compile (nlo_setup, pn, global)
  type(cmd_nlo_setup_t), pointer :: nlo_setup
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_id, pn_options
  allocate (nlo_setup)
  call rt_data_local_init (nlo_setup%local, global)
  pn_id => parse_node_get_sub_ptr (pn, 2)
  pn_options => parse_node_get_next_ptr (pn_id)
  nlo_setup%id = parse_node_get_string (pn_id)
  if (associated (pn_options)) then
    allocate (nlo_setup%options)
    call command_list_compile (nlo_setup%options, pn_options, &
      nlo_setup%local)
  end if
end subroutine cmd_nlo_setup_compile

```

Helper for handling dipole-related variables

{CCC Commands: procedures}+≡

```

subroutine handle_nlo_variables (global)
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: recombination_mode
  if (var_list_is_known (global%var_list, "?dipole_resolve_emission")) &
    call nlo_setup_list_append (global%nlo_setup_list, &
      NLO_SETUP_SET_RESOLVE, resolve = &
        var_list_get_lval (global%var_list, "?dipole_resolve_emission"))
  if (var_list_is_known (global%var_list, "?recombination_complement")) &
    call nlo_setup_list_append (global%nlo_setup_list, &
      NLO_SETUP_SET_RECOMBINATION_COMPLEMENT, recombination_complement = &
        var_list_get_lval (global%var_list, "?recombination_complement"))
  if (var_list_is_known (global%var_list, "recombination_scale")) &
    call nlo_setup_list_append (global%nlo_setup_list, &
      NLO_SETUP_SET_MRECOMB, mrecomb = &
        var_list_get_rval (global%var_list, "recombination_scale"))
  if (var_list_is_known (global%var_list, "photon_beam_separation")) &
    call nlo_setup_list_append (global%nlo_setup_list, &
      NLO_SETUP_SET_PHOTON_BEAM_SEPARATION, photon_beam_separation = &
        var_list_get_rval (global%var_list, "photon_beam_separation"))
  if (var_list_is_known (global%var_list, "$recombination_mode")) then
    recombination_mode = var_list_get_sval (global%var_list, &
      "$recombination_mode")
    select case (char (recombination_mode))
      case ("raccoon")
        call nlo_setup_list_append (global%nlo_setup_list, &
          NLO_SETUP_SET_RECOMBINATION, &
            recombination = NLO_RECOMBINATION_RACCOON)
      case ("ignore")
        call nlo_setup_list_append (global%nlo_setup_list, &
          NLO_SETUP_SET_RECOMBINATION, &
            recombination = NLO_RECOMBINATION_IGNORE_PHOTON)
      case ("barbara_ww")
        call nlo_setup_list_append (global%nlo_setup_list, &
          NLO_SETUP_SET_RECOMBINATION, &

```

```

        recombination = NLO_RECOMBINATION_BARBARA_WW)
    case default
        call msg_error ("unknown recombination mode '" // &
            char (recombination_mode) // "'")
    end select
end if
end subroutine handle_nlo_variables

```

*<CCC Commands: procedures>+≡*

```

subroutine cmd_nlo_setup_execute (nlo_setup, global)
    type(cmd_nlo_setup_t), intent(inout), target :: nlo_setup
    type(rt_data_t), intent(inout), target :: global
    integer :: ci_type
    call rt_data_link (nlo_setup%local, global)
    if (associated (nlo_setup%options)) call command_list_execute ( &
        nlo_setup%options, nlo_setup%local)
    call handle_nlo_variables (nlo_setup%local)
    ci_type = process_library_get_ci_type (nlo_setup%local%prc_lib, &
        nlo_setup%id)
    if (ci_type < 0) then
        call msg_error ("process " // char (nlo_setup%id) &
            // " not found in library; skipping...")
    else
        call process_library_apply_nlo_setup ( &
            nlo_setup%local%prc_lib, nlo_setup%id, &
            nlo_setup%local%nlo_setup_list)
    end if
    call rt_data_restore (global, nlo_setup%local)
end subroutine cmd_nlo_setup_execute

```

## Process configuration

We define a process-configuration command as a specific type. The incoming and outgoing particles are given evaluation-trees which we transform to PDG-code arrays. For transferring to O'MEGA, they are reconverted to strings.

This also includes the choice of method for the corresponding process: **omega** for O'MEGA matrix elements, **test** for special processes generated by WHIZARD, **external** for using an external matrix element and **dipole** for generating dipole subtraction terms.

*<Commands: types>+≡*

```

type, extends (command_t) :: cmd_process_t
    private
    type(string_t) :: id
    integer :: n_in = 0
    integer :: n_out = 0
    type(parse_node_p), dimension(:), allocatable :: pn_pdg_in
    type(parse_node_p), dimension(:), allocatable :: pn_pdg_out
!    type(command_list_t), pointer :: options => null ()
    type(rt_data_t), pointer :: local
contains
    <Commands: cmd process: TBP>
end type cmd_process_t

```

Finalize. Delete the options list.

```

<CCC Commands: cmd process: TBP>≡
  procedure :: final => cmd_process_final

<CCC Commands: procedures>+≡
  subroutine cmd_process_final (cmd)
    class(cmd_process_t), intent(inout) :: cmd
    if (associated (cmd%options)) then
      call command_list_final (cmd%options)
      deallocate (cmd%options)
    end if
  end subroutine cmd_process_final

```

Output. The particle expressions are not resolved, so we just list the number of incoming and outgoing particles.

```

<Commands: cmd process: TBP>≡
  procedure :: write => cmd_process_write

<Commands: procedures>+≡
  subroutine cmd_process_write (cmd, unit, indent)
    class(cmd_process_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    write (u, "(1x,A,A,A,I0,A,I0,A)") "process: ", char (cmd%id), " (", &
      size (cmd%pn_pdg_in), " -> ", size (cmd%pn_pdg_out), ")"
  end subroutine cmd_process_write

```

Compile. Find and assign the parse nodes.

Note: the local environment is identified with the global one. We should keep a separate local environment.

```

<Commands: cmd process: TBP>+≡
  procedure :: compile => cmd_process_compile

<Commands: procedures>+≡
  subroutine cmd_process_compile (cmd, global)
    class(cmd_process_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_id, pn_in, pn_out, pn_codes, pn_opt
    integer :: i
    pn_id => parse_node_get_sub_ptr (cmd%pn, 2)
    pn_in => parse_node_get_next_ptr (pn_id, 2)
    pn_out => parse_node_get_next_ptr (pn_in, 2)
    pn_opt => parse_node_get_next_ptr (pn_out)
    ! call rt_data_local_init (cmd%local, global)
    ! if (associated (pn_opt)) then
    !   allocate (cmd%options)
    !   call command_list_compile (cmd%options, pn_opt, cmd%local)
    ! end if
    cmd%local => global
    cmd%id = parse_node_get_string (pn_id)

```

```

!      call process_library_check_name_consistency (cmd%id, global%prc_lib)
cmd%n_in  = parse_node_get_n_sub (pn_in)
cmd%n_out = parse_node_get_n_sub (pn_out)
pn_codes => parse_node_get_sub_ptr (pn_in)
allocate (cmd%pn_pdg_in (cmd%n_in))
do i = 1, cmd%n_in
    cmd%pn_pdg_in(i)%ptr => pn_codes
    pn_codes => parse_node_get_next_ptr (pn_codes)
end do
pn_codes => parse_node_get_sub_ptr (pn_out)
allocate (cmd%pn_pdg_out (cmd%n_out))
do i = 1, cmd%n_out
    cmd%pn_pdg_out(i)%ptr => pn_codes
    pn_codes => parse_node_get_next_ptr (pn_codes)
end do
!      call var_list_init_num_id (global%var_list, cmd%id)
!      call rt_data_local_reset (cmd%local)
end subroutine cmd_process_compile

```

Command execution. Evaluate the subevents, transform PDG codes into strings, and add the current process configuration to the process library. If anything goes wrong in the particle code interpretation, the particle names will contain question marks. In that case, the process is not recorded.

*(Commands: cmd process: TBP)+≡*

```

procedure :: execute => cmd_process_execute

```

*(Commands: procedures)+≡*

```

subroutine cmd_process_execute (cmd, global)
class(cmd_process_t), intent(inout) :: cmd
type(rt_data_t), intent(inout), target :: global
type(pdg_array_t) :: pdg_in, pdg_out
type(string_t), dimension(:), allocatable :: prt_in
type(string_t), dimension(:), allocatable :: prt_out
type(process_configuration_t) :: prc_config
integer :: i

!      integer :: method
!      logical :: rebuild_library, omega_openmp
!      type(string_t) :: restrictions, omega_flags, method_str
!      integer :: dipole_flavor
!      type(nlo_setup_t) :: nlo_setup
!      logical :: broken_declaration
!      type(flavor_t), dimension(:), allocatable :: flavors
!      integer :: ci_type
!      call rt_data_link (cmd%local, global)
!      if (associated (cmd%options)) then
!          call command_list_execute (cmd%options, cmd%local)
!      end if
!      call handle_nlo_variables (cmd%local)
!      if (process_library_is_static (global%prc_lib)) then
!          call msg_error ("Process library '" &
!              // char (process_library_get_name (global%prc_lib)) &
!              // "' is static, no processes can be added")
!      return

```



```

!       end if
allocate (prt_in (size (cmd%pn_pdg_in)))
do i = 1, size (cmd%pn_pdg_in)
    pdg_in = &
        eval_pdg_array (cmd%pn_pdg_in(i)%ptr, cmd%local%var_list)
    prt_in(i) = make_flavor_string (pdg_in, cmd%local%model)
end do
allocate (prt_out (size (cmd%pn_pdg_out)))
do i = 1, size (cmd%pn_pdg_out)
    pdg_out = &
        eval_pdg_array (cmd%pn_pdg_out(i)%ptr, cmd%local%var_list)
    prt_out(i) = make_flavor_string (pdg_out, cmd%local%model)
end do
!       restrictions = var_list_get_sval &
!           (cmd%local%var_list, var_str ("restrictions"))
!       omega_flags = var_list_get_sval &
!           (cmd%local%var_list, var_str ("omega_flags"))
!       method_str = var_list_get_sval &
!           (cmd%local%var_list, var_str ("method"))
!       omega_openmp = var_list_get_lval &
!           (cmd%local%var_list, var_str ("?omega_openmp"))
!       method = method_of_string (method_str)

call prc_config%init (cmd%id, size (prt_in), 1, global)
call prc_config%setup_component (1, &
    new_prt_spec (prt_in), new_prt_spec (prt_out), global)
call prc_config%record (global)

!       broken_declaration = .false.
!       if (all (scan (cmd%prt_in, "?") > 0) .or. &
!           all (scan (cmd%prt_out, "?") > 0)) then
!           broken_declaration = .true.
!           call msg_error ("Broken process declaration: skipped")
!       else
!           select case (char (method_str))
!           case ("dipole_integrated_qed")
!               ci_type = CI_DIPOLE_INTEGRATED_QED
!               if (size (cmd%prt_in) < 2) then
!                   broken_declaration = .true.
!                   call msg_error ("dipoles are not available for decays")
!               else
!                   call nlo_setup_init (nlo_setup, &
!                       size (cmd%prt_in) + size (cmd%prt_out))
!                   call nlo_setup_apply_list (nlo_setup, &
!                       cmd%local%nlo_setup_list)
!               end if
!           case ("dipole_real_qed")
!               ci_type = CI_DIPOLE_REAL_QED
!               if (size (cmd%prt_in) < 2) then
!                   broken_declaration = .true.
!                   call msg_error ("dipoles are not available for decays")
!               else
!                   call nlo_setup_init (nlo_setup, &
!                       size (cmd%prt_in) + size (cmd%prt_out))

```

```

!           call nlo_setup_apply_list (nlo_setup, &
!             cmd%local%nlo_setup_list)
!         end if
!       case ("photon_recombination")
!         ci_type = CI_PHOTON_RECOMBINATION
!         if (size (cmd%prt_in) < 2) then
!           broken_declaration = .true.
!           call msg_error ("photon recombination is not available for" &
!             // "decays")
!         else
!           call nlo_setup_init (nlo_setup, &
!             size (cmd%prt_in) + size (cmd%prt_out))
!           call nlo_setup_apply_list (nlo_setup, &
!             cmd%local%nlo_setup_list)
!         end if
!       case default
!         ci_type = CI_OMEGA
!       end select
!     end if
!   if (.not. broken_declaration) then
!     rebuild_library = var_list_get_lval &
!       (cmd%local%var_list, var_str ("?rebuild_library"))
!     call process_library_append (global%prc_lib, ci_type, &
!       cmd%id, cmd%local%model, &
!       cmd%prt_in, cmd%prt_out, &
!       method = method, restrictions = restrictions, &
!       omega_flags = omega_flags, &
!       rebuild_library = rebuild_library, message = .true., &
!       omega_openmp = omega_openmp, &
!       nlo_setup=nlo_setup)
!   else
!     call msg_error ("Broken process declaration: skipped")
!   end if
!   call var_list_init_num_id (global%var_list, cmd%id)
!   call rt_data_restore (global, cmd%local)
end subroutine cmd_process_execute

```

This is a method of the eval tree, but cannot be coded inside the `expressions` module since it uses the `model` and `flv` types which are not available there.

(*Commands: procedures*) +=

```

function make_flavor_string (aval, model) result (prt)
  type(string_t) :: prt
  type(pdg_array_t), intent(in) :: aval
  type(model_t), intent(in), target :: model
  integer, dimension(:), allocatable :: pdg
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  pdg = aval
  allocate (flv (size (pdg)))
  call flavor_init (flv, pdg, model)
  if (size (pdg) /= 0) then
    prt = flavor_get_name (flv(1))
    do i = 2, size (flv)

```

```

        prt = prt // ":" // flavor_get_name (flv(i))
    end do
else
    prt = "?"
end if
end function make_flavor_string

```

Auxiliary function to convert the method string into an integer identifier.

```

<CCC Commands: procedures>+≡
function method_of_string (meth_str) result (meth_id)
    type(string_t), intent(in) :: meth_str
    integer :: meth_id
    select case (char (meth_str))
    case ("omega", "dipole_integrated_qed", "dipole_real_qed", &
        "dipole_integrated_qcd", "dipole_real_qcd", "photon_recombination")
        meth_id = PRC_OMEGA
    case ("test")
        meth_id = PRC_TEST
    case ("unit")
        meth_id = PRC_UNIT
    case ("external")
        meth_id = PRC_EXTERNAL
    case default
        call msg_fatal ("Invalid method for matrix elements.")
    end select
end function method_of_string

```

## Process compilation

```

<Commands: types>+≡
type, extends (command_t) :: cmd_compile_t
    private
    type(string_t), dimension(:), allocatable :: libname
    logical :: make_executable = .false.
    type(string_t) :: exec_name
!    type(command_list_t), pointer :: options => null ()
    type(rt_data_t), pointer :: local
contains
    <Commands: cmd compile: TBP>
end type cmd_compile_t

```

Finalize. Only options.

```

<CCC Commands: cmd compile: TBP>≡
procedure :: final => cmd_compile_final

<CCC Commands: procedures>+≡
subroutine cmd_compile_final (cmd)
    class(cmd_compile_t), intent(inout) :: cmd
    if (associated (cmd%options)) then
        call command_list_final (cmd%options)
        deallocate (cmd%options)
    end if

```

```
end subroutine cmd_compile_final
```

Output: list all libraries to be compiled.

*<Commands: cmd compile: TBP>≡*

```
procedure :: write => cmd_compile_write
```

*<Commands: procedures>+≡*

```
subroutine cmd_compile_write (cmd, unit, indent)
  class(cmd_compile_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "compile ("
  if (allocated(cmd%libname)) then
    do i = 1, size (cmd%libname)
      if (i > 1) write (u, "(A,1x)", advance="no") " ,"
      write (u, "('\"',A,'\"')", advance="no") char (cmd%libname(i))
    end do
  end if
  write (u, "(A)") ")"
end subroutine cmd_compile_write
```

Compile the libraries specified in the argument. If the argument is empty, compile all libraries which can be found in the process library stack.

*<Commands: cmd compile: TBP>+≡*

```
procedure :: compile => cmd_compile_compile
```

*<Commands: procedures>+≡*

```
subroutine cmd_compile_compile (cmd, global)
  class(cmd_compile_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_arg, pn_lib, pn_opt
  type(parse_node_t), pointer :: pn_exec_name_spec, pn_exec_name
  integer :: n_lib, i
  pn_cmd => parse_node_get_sub_ptr (cmd%pn)
  pn_clause => parse_node_get_sub_ptr (pn_cmd)
  pn_exec_name_spec => parse_node_get_sub_ptr (pn_clause, 2)
  if (associated (pn_exec_name_spec)) then
    pn_exec_name => parse_node_get_sub_ptr (pn_exec_name_spec, 2)
  else
    pn_exec_name => null ()
  end if
  pn_arg => parse_node_get_next_ptr (pn_clause)
  pn_opt => parse_node_get_next_ptr (pn_cmd)
!   call rt_data_local_init (cmd%local, global)
  cmd%local => global
!   if (associated (pn_opt)) then
!     allocate (cmd%options)
!     call command_list_compile (cmd%options, pn_opt, cmd%local)
!   end if
  if (associated (pn_arg)) then
    n_lib = parse_node_get_n_sub (pn_arg)
  else
```

```

        n_lib = 0
    end if
    if (n_lib > 0) then
        allocate (cmd%libname (n_lib))
        pn_lib => parse_node_get_sub_ptr (pn_arg)
        do i = 1, n_lib
            cmd%libname(i) = parse_node_get_string (pn_lib)
            pn_lib => parse_node_get_next_ptr (pn_lib)
        end do
    end if
    if (associated (pn_exec_name)) then
        call msg_bug ("Static executable building temporarily disabled")
        cmd%make_executable = .true.
        cmd%exec_name = parse_node_get_string (pn_exec_name)
    end if
!    call rt_data_local_reset (cmd%local)
end subroutine cmd_compile_compile

```

Command execution. Generate code, write driver, compile and link. Do this for all libraries in the list.

If no library names have been given and stored while compiling this command, we collect all libraries from the current stack and compile those.

*<Commands: cmd compile: TBP>+≡*

```

    procedure :: execute => cmd_compile_execute

```

*<Commands: procedures>+≡*

```

    subroutine cmd_compile_execute (cmd, global)
        class(cmd_compile_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(string_t), dimension(:), allocatable :: libname
        integer :: i
!        call rt_data_link (cmd%local, global)
!        if (associated (cmd%options)) then
!            call command_list_execute (cmd%options, cmd%local)
!        end if
        if (allocated (cmd%libname)) then
            allocate (libname (size (cmd%libname)))
            libname = cmd%libname
        else
            call cmd%local%prclib_stack%get_names (libname)
        end if
        if (cmd%make_executable) then
!            call compile_executable &
!                (cmd%libname, cmd%exec_name, cmd%local)
        else
            do i = 1, size (libname)
                call compile_library (libname(i), cmd%local)
!                (cmd%libname, cmd%local, global%var_list, global%prc_lib)
            end do
        end if
!        call rt_data_restore (global, cmd%local)
    end subroutine cmd_compile_execute

```

## Process sum

The little brother of process configuration.

```
<CCC Commands: types>+≡
  type :: cmd_process_sum_t
    private
    type(string_t) :: id, child1, child2
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
  end type
```

Finalize.

```
<CCC Commands: procedures>+≡
  subroutine cmd_process_sum_final (prc_sum)
    type(cmd_process_sum_t), intent(inout) :: prc_sum
    if (associated (prc_sum%options)) &
      call command_list_final (prc_sum%options)
  end subroutine cmd_process_sum_final
```

Compile.

```
<CCC Commands: procedures>+≡
  subroutine cmd_process_sum_compile (prc_sum, pn, global)
    type(cmd_process_sum_t), intent(inout), pointer :: prc_sum
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_id, pn_child1, pn_child2, pn_opt
    allocate (prc_sum)
    pn_id => parse_node_get_sub_ptr (pn, 2)
    pn_child1 => parse_node_get_next_ptr (pn_id, 2)
    pn_child2 => parse_node_get_next_ptr (pn_child1, 2)
    pn_opt => parse_node_get_next_ptr (pn_child2)
    prc_sum%id = parse_node_get_string (pn_id)
    prc_sum%child1 = parse_node_get_string (pn_child1)
    prc_sum%child2 = parse_node_get_string (pn_child2)
    call rt_data_local_init (prc_sum%local, global)
    if (associated (pn_opt)) then
      allocate (prc_sum%options)
      call command_list_compile (prc_sum%options, pn_opt, prc_sum%local)
    end if
    call var_list_init_num_id (global%var_list, prc_sum%id)
    call rt_data_local_reset (prc_sum%local)
  end subroutine cmd_process_sum_compile
```

Execute.

```
<CCC Commands: procedures>+≡
  subroutine cmd_process_sum_execute (prc_sum, global)
    type(cmd_process_sum_t), intent(inout), target :: prc_sum
    type(rt_data_t), intent(inout), target :: global
    type(string_t) :: child1, child2
    type(nlo_setup_t) :: nlo_setup
    call rt_data_link (prc_sum%local, global)
    if (associated (prc_sum%options)) &
```

```

        call command_list_execute (prc_sum%options, prc_sum%local)
    call handle_nlo_variables (prc_sum%local)
    call nlo_setup_init (nlo_setup)
    call nlo_setup_apply_list (nlo_setup, prc_sum%local%nlo_setup_list)
    call process_library_append_prc_sum ( &
        global%prc_lib, prc_sum%id, prc_sum%child1, prc_sum%child2, &
        nlo_setup, message = .true.)
    call rt_data_restore (global, prc_sum%local)
end subroutine cmd_process_sum_execute

```

## Load library

For the most part, this is analogous to the compile command.

```

<CCC Commands: types>+≡
    type :: cmd_load_t
        private
        type(string_t), dimension(:), allocatable :: libname
        type(command_list_t), pointer :: options => null ()
        type(rt_data_t) :: local
    end type cmd_load_t

```

Finalize. Only options.

```

<CCC Commands: procedures>+≡
    subroutine cmd_load_final (load)
        type(cmd_load_t), intent(inout) :: load
        if (associated (load%options)) then
            call command_list_final (load%options)
            deallocate (load%options)
        end if
    end subroutine cmd_load_final

```

Compile the load command.

```

<CCC Commands: procedures>+≡
    subroutine cmd_load_compile (load, pn, global)
        type(cmd_load_t), pointer :: load
        type(parse_node_t), intent(in), target :: pn
        type(rt_data_t), intent(in), target :: global
        type(parse_node_t), pointer :: pn_arg, pn_lib, pn_opt
        type(process_library_t), pointer :: prc_lib
        integer :: n_lib, i
        pn_arg => parse_node_get_sub_ptr (pn, 2)
        allocate (load)
        if (associated (pn_arg)) then
            select case (char (parse_node_get_rule_key (pn_arg)))
            case ("load_arg")
                n_lib = parse_node_get_n_sub (pn_arg)
                pn_opt => parse_node_get_next_ptr (pn_arg)
            case default
                n_lib = 0
                pn_opt => pn_arg
            end select
        end if
    end subroutine cmd_load_compile

```

```

else
    n_lib = 0
    pn_opt => null ()
end if
call rt_data_local_init (load%local, global)
if (associated (pn_opt)) then
    allocate (load%options)
    call command_list_compile (load%options, pn_opt, load%local)
end if
if (n_lib > 0) then
    allocate (load%libname (n_lib))
    pn_lib => parse_node_get_sub_ptr (pn_arg)
    do i = 1, n_lib
        load%libname(i) = parse_node_get_string (pn_lib)
        pn_lib => parse_node_get_next_ptr (pn_lib)
    end do
else
    n_lib = 0
    prc_lib => process_library_store_get_first ()
    do while (associated (prc_lib))
        n_lib = n_lib + 1
        call process_library_advance (prc_lib)
    end do
    allocate (load%libname (n_lib))
    i = 0
    prc_lib => process_library_store_get_first ()
    do while (associated (prc_lib))
        i = i + 1
        load%libname(i) = process_library_get_name (prc_lib)
        call process_library_advance (prc_lib)
    end do
end if
call rt_data_local_reset (load%local)
end subroutine cmd_load_compile

```

Execute: Load the specified shared libraries. Insert a pointer to the last library in the list into the global record.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_load_execute (load, global)
    type(cmd_load_t), intent(inout) :: load
    type(rt_data_t), intent(inout), target :: global
    call rt_data_link (load%local, global)
    if (associated (load%options)) then
        call command_list_execute (load%options, load%local)
    end if
    call load_library &
        (load%libname, load%local, global%var_list, global%prc_lib)
    call rt_data_restore (global, load%local)
end subroutine cmd_load_execute

```



## Execute a shell command

The argument is a string expression.

```
<CCC Commands: types>+≡
  type :: cmd_exec_t
    private
    type(parse_node_t), pointer :: pn_command => null ()
  end type cmd_exec_t
```

Delete the eval tree.

```
<CCC Commands: procedures>+≡
  subroutine cmd_exec_final (exec)
    type(cmd_exec_t), intent(inout) :: exec
  end subroutine cmd_exec_final
```

Compile the exec command.

```
<CCC Commands: procedures>+≡
  subroutine cmd_exec_compile (exec, pn, global)
    type(cmd_exec_t), pointer :: exec
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_command
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    pn_command => parse_node_get_sub_ptr (pn_arg)
    allocate (exec)
    exec%pn_command => pn_command
  end subroutine cmd_exec_compile
```

Execute the specified shell command.

```
<CCC Commands: procedures>+≡
  subroutine cmd_exec_execute (exec, global)
    type(cmd_exec_t), intent(inout) :: exec
    type(rt_data_t), intent(in) :: global
    type(string_t) :: command
    logical :: is_known
    integer :: status
    command = eval_string (exec%pn_command, global%var_list, is_known=is_known)
    if (is_known) then
      if (command /= "") then
        call os_system_call (command, status, verbose=.true.)
        if (status /= 0) then
          write (msg_buffer, "(A,I0)") "Return code = ", status
          call msg_message ()
          call msg_error ("System command returned with nonzero status code")
        end if
      end if
    end if
  end subroutine cmd_exec_execute
```

## Variable declaration

A variable can have various types. Hold the definition as an eval tree.

```
<Commands: types>+≡
  type, extends (command_t) :: cmd_var_t
    private
      type(string_t) :: name
      integer :: type = V_NONE
      type(parse_node_t), pointer :: pn_value => null ()
      logical :: is_intrinsic = .false.
      logical :: is_copy = .false.
    contains
      <Commands: cmd var: TBP>
    end type cmd_var_t
```

Delete the eval tree.

```
<CCC Commands: procedures>+≡
  subroutine cmd_var_final (var)
    type(cmd_var_t), intent(inout) :: var
  end subroutine cmd_var_final
```

Output. We know name, type, and properties, but not the value.

```
<Commands: cmd var: TBP>≡
  procedure :: write => cmd_var_write
<Commands: procedures>+≡
  subroutine cmd_var_write (cmd, unit, indent)
    class(cmd_var_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u, i
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    write (u, "(1x,A,A,A)", advance="no") "var: ", char (cmd%name), " ("
    select case (cmd%type)
    case (V_NONE)
      write (u, "(A)", advance="no") "[unknown]"
    case (V_LOG)
      write (u, "(A)", advance="no") "logical"
    case (V_INT)
      write (u, "(A)", advance="no") "int"
    case (V_REAL)
      write (u, "(A)", advance="no") "real"
    case (V_CMPLX)
      write (u, "(A)", advance="no") "complex"
    case (V_STR)
      write (u, "(A)", advance="no") "string"
    case (V_PDG)
      write (u, "(A)", advance="no") "alias"
    end select
    if (cmd%is_intrinsic) then
      write (u, "(A)", advance="no") ", intrinsic"
    end if
    if (cmd%is_copy) then
      write (u, "(A)", advance="no") ", copy"
```

```

end if
write (u, "(A)" )"
end subroutine cmd_var_write

```

Compile the lhs and determine the variable name and type. Check whether this variable can be created or modified as requested, and append the value to the variable list, if appropriate. The value is initially undefined. The rhs is assigned to a pointer, to be compiled and evaluated when the command is executed.

*(Commands: cmd var: TBP)+≡*

```

procedure :: compile => cmd_var_compile

```

*(Commands: procedures)+≡*

```

subroutine cmd_var_compile (cmd, global)
  class(cmd_var_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_var, pn_name
  type(parse_node_t), pointer :: pn_result, pn_proc
  type(string_t) :: var_name
  type(var_entry_t), pointer :: var_entry
  integer :: type
  logical :: new
  pn_result => null ()
  new = .false.
  select case (char (parse_node_get_rule_key (cmd%pn)))
  case ("cmd_log_decl");      type = V_LOG
    pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
    if (.not. associated (pn_var)) then    ! handle masked syntax error
      cmd%type = V_NONE; return
    end if
    pn_name => parse_node_get_sub_ptr (pn_var, 2)
    new = .true.
  case ("cmd_log");           type = V_LOG
    pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
  case ("cmd_int");           type = V_INT
    pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
    new = .true.
  case ("cmd_real");          type = V_REAL
    pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
    new = .true.
  case ("cmd_complex");       type = V_CMPLX
    pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
    new = .true.
  case ("cmd_num");           type = V_NONE
    pn_name => parse_node_get_sub_ptr (cmd%pn)
  case ("cmd_string_decl");   type = V_STR
    pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
    if (.not. associated (pn_var)) then    ! handle masked syntax error
      cmd%type = V_NONE; return
    end if
    pn_name => parse_node_get_sub_ptr (pn_var, 2)
    new = .true.
  case ("cmd_string");        type = V_STR
    pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
  case ("cmd_alias");         type = V_PDG

```

```

        pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
        new = .true.
case ("cmd_result");          type = V_REAL
    pn_name => parse_node_get_sub_ptr (cmd%pn)
    pn_result => parse_node_get_sub_ptr (pn_name)
    pn_proc => parse_node_get_next_ptr (pn_result)
case default
    call parse_node_mismatch &
        ("logical|int|real|complex|?|${alias}|var_name", cmd%pn) ! $
end select
if (.not. associated (pn_name)) then    ! handle masked syntax error
    cmd%type = V_NONE; return
end if
if (.not. associated (pn_result)) then
    var_name = parse_node_get_string (pn_name)
else
    var_name = parse_node_get_key (pn_result) &
        // "(" // parse_node_get_string (pn_proc) // ")"
end if
select case (type)
case (V_LOG); var_name = "?" // var_name
case (V_STR); var_name = "$" // var_name    ! $
end select
call var_list_check_user_var (global%var_list, var_name, type, new)
cmd%type = type
cmd%name = var_name
var_entry => var_list_get_var_ptr &
    (global%var_list, cmd%name, cmd%type, follow_link=.false.)
cmd%pn_value => parse_node_get_next_ptr (pn_name, 2)
if (.not. associated (cmd%pn_value)) cmd%type = V_NONE
if (associated (var_entry)) then
    cmd%is_intrinsic = var_entry_is_intrinsic (var_entry)
    cmd%is_copy = var_entry_is_copy (var_entry)
else
    var_entry => var_list_get_var_ptr &
        (global%var_list, cmd%name, cmd%type, follow_link=.true.)
    if (associated (var_entry)) then
        cmd%is_intrinsic = var_entry_is_intrinsic (var_entry)
        if (var_entry_is_copy (var_entry)) then
            cmd%is_copy = .true.
            call var_list_init_copy (global%var_list, var_entry, user=.true.)
        end if
    end if
    if (.not. cmd%is_copy) then
        select case (cmd%type)
        case (V_LOG)
            call var_list_append_log (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)
        case (V_INT)
            call var_list_append_int (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)
        case (V_REAL)
            call var_list_append_real (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)

```

```

        case (V_CMPLX)
            call var_list_append_cmplx (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)
        case (V_PDG)
            call var_list_append_pdg_array (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)
        case (V_STR)
            call var_list_append_string (global%var_list, cmd%name, &
                intrinsic=cmd%is_intrinsic, user=.true.)
        end select
    end if
end if
end subroutine cmd_var_compile

```

Execute. Evaluate the definition and assign the variable value. If the variable is a copy, the original is a model variable. The original is set automatically, and an update of the dependent parameters is in order.

```

<Commands: cmd var: TBP>+≡
    procedure :: execute => cmd_var_execute

<Commands: procedures>+≡
    subroutine cmd_var_execute (cmd, global)
        class(cmd_var_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(string_t) :: model_name
        type(var_list_t), pointer :: model_vars
        if (associated (global%model)) then
            model_name = model_get_name (global%model)
            model_vars => model_get_var_list_ptr (global%model)
            if (cmd%is_copy) then
                call var_list_set_original_pointer (global%var_list, cmd%name, &
                    model_vars)
            end if
            call cmd%set_value (global%var_list, &
                verbose=.true., model_name=model_name)
            if (cmd%is_copy) then
                call model_parameters_update (global%model)
                call var_list_synchronize (global%var_list, model_vars)
            end if
        else
            call cmd_var_set_value (cmd, global%var_list, verbose=.true.)
        end if
    end subroutine cmd_var_execute

```

Copy the value to the variable list, where the variable should already exist.

```

<Commands: cmd var: TBP>+≡
    procedure :: set_value => cmd_var_set_value

<Commands: procedures>+≡
    subroutine cmd_var_set_value (var, var_list, verbose, model_name)
        class(cmd_var_t), intent(inout) :: var
        type(var_list_t), intent(inout), target :: var_list
        logical, intent(in), optional :: verbose
        type(string_t), intent(in), optional :: model_name

```

```

logical :: lval
integer :: ival
real(default) :: rval
complex(default) :: cval
type(pdg_array_t) :: aval
type(string_t) :: sval
logical :: is_known
select case (var%type)
case (V_LOG)
    lval = eval_log (var%pn_value, var_list, is_known=is_known)
    call var_list_set_log (var_list, var%name, &
        lval, is_known, verbose=verbose, model_name=model_name)
case (V_INT)
    ival = eval_int (var%pn_value, var_list, is_known=is_known)
    call var_list_set_int (var_list, var%name, &
        ival, is_known, verbose=verbose, model_name=model_name)
case (V_REAL)
    rval = eval_real (var%pn_value, var_list, is_known=is_known)
    call var_list_set_real (var_list, var%name, &
        rval, is_known, verbose=verbose, model_name=model_name)
case (V_CMPLX)
    cval = eval_cmplx (var%pn_value, var_list, is_known=is_known)
    call var_list_set_cmplx (var_list, var%name, &
        cval, is_known, verbose=verbose, model_name=model_name)
case (V_PDG)
    aval = eval_pdg_array (var%pn_value, var_list, is_known=is_known)
    call var_list_set_pdg_array (var_list, var%name, &
        aval, is_known, verbose=verbose, model_name=model_name)
case (V_STR)
    sval = eval_string (var%pn_value, var_list, is_known=is_known)
    call var_list_set_string (var_list, var%name, &
        sval, is_known, verbose=verbose, model_name=model_name)
end select
end subroutine cmd_var_set_value

```

## SLHA

Read a SLHA (SUSY Les Houches Accord) file to fill the appropriate model parameters. We do not access the current variable record, but directly work on the appropriate SUSY model, which is loaded if necessary.

We may be in read or write mode. In the latter case, we may write just input parameters, or the complete spectrum, or the spectrum with all decays.

*<CCC Commands: types>+≡*

```

type :: cmd_slha_t
private
type(string_t) :: file
logical :: write = .false.
type(command_list_t), pointer :: options => null ()
type(rt_data_t) :: local
end type cmd_slha_t

```

Finalizer.

```
(CCC Commands: procedures) +=  
subroutine cmd_slha_final (slha)  
  type(cmd_slha_t), intent(inout) :: slha  
  if (associated (slha%options)) then  
    call command_list_final (slha%options)  
    deallocate (slha%options)  
  end if  
end subroutine cmd_slha_final
```

Compile. Read the filename and store it.

```
(CCC Commands: procedures) +=  
subroutine cmd_slha_compile (slha, pn, global)  
  type(cmd_slha_t), pointer :: slha  
  type(parse_node_t), intent(in), target :: pn  
  type(rt_data_t), intent(in), target :: global  
  type(parse_node_t), pointer :: pn_key, pn_arg, pn_file  
  type(parse_node_t), pointer :: pn_opt  
  pn_key => parse_node_get_sub_ptr (pn)  
  pn_arg => parse_node_get_next_ptr (pn_key)  
  pn_file => parse_node_get_sub_ptr (pn_arg)  
  pn_opt => parse_node_get_next_ptr (pn_arg)  
  allocate (slha)  
  call rt_data_local_init (slha%local, global)  
  if (associated (pn_opt)) then  
    allocate (slha%options)  
    call command_list_compile (slha%options, pn_opt, slha%local)  
  end if  
  select case (char (parse_node_get_key (pn_key)))  
  case ("read_slha")  
    slha%write = .false.  
  case ("write_slha")  
    slha%write = .true.  
  case default  
    call parse_node_mismatch ("read_slha|write_slha", pn)  
  end select  
  slha%file = parse_node_get_string (pn_file)  
  call rt_data_local_reset (slha%local)  
end subroutine cmd_slha_compile
```

Execute. Read or write the specified SLHA file.

```
(CCC Commands: procedures) +=  
subroutine cmd_slha_execute (slha, global)  
  type(cmd_slha_t), intent(inout), target :: slha  
  type(rt_data_t), intent(inout), target :: global  
  logical :: input, spectrum, decays  
  call rt_data_link (slha%local, global)  
  if (associated (slha%options)) then  
    call command_list_execute (slha%options, slha%local)  
  end if  
  if (slha%write) then  
    input = .true.  
    spectrum = .false.
```

```

        decays = .false.
        call slha_write_file &
            (slha%file, slha%local%model, &
             input = input, spectrum = spectrum, decays = decays)
    else
        input = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_input"))
        spectrum = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_spectrum"))
        decays = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_decays"))
        call slha_read_file &
            (slha%file, slha%local%os_data, slha%local%model, &
             input = input, spectrum = spectrum, decays = decays)
    end if
    call rt_data_restore (global, slha%local, keep_model_vars = .true.)
end subroutine cmd_slha_execute

```

### Show values of variables

```

<CCC Commands: types>+≡
type :: cmd_show_t
    private
    type(string_t), dimension(:), allocatable :: name
    type(parse_node_p), dimension(:), allocatable :: pn_expr
    type(var_entry_t), dimension(:), allocatable :: value
end type cmd_show_t

```

Finalize.

```

<CCC Commands: procedures>+≡
subroutine cmd_show_final (show)
    type(cmd_show_t), intent(inout) :: show
    integer :: i
    if (allocated (show%pn_expr)) then
        deallocate (show%pn_expr)
    end if
    if (allocated (show%value)) then
        do i = 1, size (show%value)
            call var_entry_final (show%value(i))
        end do
    end if
end subroutine cmd_show_final

```

Compile. Allocate an array which is filled with the names of the variables to show.

```

<CCC Commands: procedures>+≡
subroutine cmd_show_compile (show, pn, global)
    type(cmd_show_t), pointer :: show
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name

```



```

type(string_t) :: key
integer :: i, n_args
pn_arg => parse_node_get_sub_ptr (pn, 2)
allocate (show)
if (associated (pn_arg)) then
  n_args = parse_node_get_n_sub (pn_arg)
  allocate (show%name (n_args))
  allocate (show%pn_expr (n_args))
  allocate (show%value (n_args))
  pn_var => parse_node_get_sub_ptr (pn_arg)
  i = 0
  do while (associated (pn_var))
    i = i + 1
    select case (char (parse_node_get_rule_key (pn_var)))
    case ("model", "beams", "results", "unstable", &
          "real", "int", "intrinsic", &
          "cuts", "scale", "factorization_scale", &
          "renormalization_scale", "weight", &
          "selection", "reweight", "analysis", "expect", "alpha_qed")
      show%name(i) = parse_node_get_key (pn_var)
    case ("library_spec")
      pn_prefix => parse_node_get_sub_ptr (pn_var)
      pn_name => parse_node_get_next_ptr (pn_prefix)
      key = parse_node_get_key (pn_prefix)
      if (associated (pn_name)) then
        show%name(i) = "L " // parse_node_get_string (pn_name)
      else
        show%name(i) = key
      end if
    case ("result_var")
      pn_prefix => parse_node_get_sub_ptr (pn_var)
      pn_name => parse_node_get_next_ptr (pn_prefix)
      if (associated (pn_name)) then
        show%name(i) = parse_node_get_key (pn_prefix) &
          // "(" // parse_node_get_string (pn_name) // ")"
      else
        show%name(i) = parse_node_get_key (pn_prefix)
      end if
    case ("log_var", "alias_var", "string_var")
      pn_prefix => parse_node_get_sub_ptr (pn_var)
      pn_name => parse_node_get_next_ptr (pn_prefix)
      key = parse_node_get_key (pn_prefix)
      if (associated (pn_name)) then
        select case (char (parse_node_get_rule_key (pn_name)))
        case ("variable")
          select case (char (key))
          case ("?", "$") ! $ sign
            show%name(i) = key // parse_node_get_string (pn_name)
          case ("alias")
            show%name(i) = "A " // parse_node_get_string (pn_name)
          case ("library")
            show%name(i) = "L " // parse_node_get_string (pn_name)
          end select
        case ("lexpr")

```

```

        show%name(i) = "<lexpr>"
        show%pn_expr(i)%ptr => pn_name
    case ("cexpr")
        show%name(i) = "<cexpr>"
        show%pn_expr(i)%ptr => pn_name
    case ("sexpr")
        show%name(i) = "<sexpr>"
        show%pn_expr(i)%ptr => pn_name
    case default
        call parse_node_mismatch &
            ("variable|expr|lexpr|cexpr|sexpr", pn_name)
    end select
else
    show%name(i) = key
end if
case ("num_var")
    pn_name => parse_node_get_sub_ptr (pn_var)
    if (associated (pn_name)) then
        select case (char (parse_node_get_rule_key (pn_name)))
        case ("variable")
            show%name(i) = parse_node_get_string (pn_name)
        case ("expr")
            show%name(i) = "<expr>"
            show%pn_expr(i)%ptr => pn_name
        case default
            call parse_node_mismatch &
                ("variable|expr", pn_name)
        end select
    else
        show%name(i) = key
    end if
case default
    pn_prefix => null ()
    pn_name => parse_node_get_sub_ptr (pn_var)
    if (associated (pn_name)) then
        show%name(i) = parse_node_get_string (pn_name)
    else
        show%name(i) = ""
    end if
end select
pn_var => parse_node_get_next_ptr (pn_var)
end do
else
    allocate (show%name (0))
end if
end subroutine cmd_show_compile

```

Execute. Scan the list of variables to show. Special cases of an empty list (show all) and keywords `int`, `real`, `alias` have to be handled as well.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_show_execute (show, global)
    type(cmd_show_t), intent(inout) :: show
    type(rt_data_t), intent(in), target :: global

```

```

type(string_t) :: name
type(process_library_t), pointer :: prc_lib
logical :: lval
integer :: ival
real(default) :: rval
complex(default) :: cval
type(pdg_array_t) :: aval
type(string_t) :: sval
logical :: is_known
integer :: result_type
integer :: i, u
u = logfile_unit ()
if (size (show%name) == 0) then
  call var_list_write (global%var_list)
else
  if (associated (global%model)) then
    name = model_get_name (global%model)
  else
    name = "[undefined]"
  end if
  do i = 1, size (show%name)
    select case (char (show%name(i)))
    case ("model")
      print *, "model* = ", char (name)
      write (u, *) "model* = ", char (name)
    case ("library")
      if (associated (global%prc_lib)) then
        call process_library_write (global%prc_lib)
        call process_library_write (global%prc_lib, unit=u)
      else
        call msg_message ("Show library: no library is loaded")
      end if
    case ("beams")
      call beam_data_write (global%beam_data)
      call beam_data_write (global%beam_data, unit=u)
    case ("results")
      call process_store_write_results ()
      call process_store_write_results (unit=u)
    case ("unstable")
      call decay_store_write ()
      call decay_store_write (unit=u)
    case ("cuts")
      if (associated (global%pn_cuts_lexpr)) then
        call parse_node_write_rec (global%pn_cuts_lexpr)
        call parse_node_write_rec (global%pn_cuts_lexpr, u)
      else
        call msg_message ("No cut expression defined")
      end if
    case ("weight")
      if (associated (global%pn_weight_expr)) then
        call parse_node_write_rec (global%pn_weight_expr)
        call parse_node_write_rec (global%pn_weight_expr, u)
      else
        call msg_message ("No weight expression defined")
      end if
    end select
  end do
end if

```

```

        end if
    case ("scale")
        if (associated (global%pn_scale_expr)) then
            call parse_node_write_rec (global%pn_scale_expr)
            call parse_node_write_rec (global%pn_scale_expr,u)
        else
            call msg_message ("No general scale expression defined")
        end if
    case ("alpha_qed")
        if (associated (global%pn_alpha_qed_expr)) then
            call parse_node_write_rec (global%pn_alpha_qed_expr)
            call parse_node_write_rec (global%pn_alpha_qed_expr, u)
        else
            call msg_message ("No expression for alpha_qed defined")
        end if
    case ("factorization_scale")
        if (associated (global%pn_fac_scale_expr)) then
            call parse_node_write_rec (global%pn_fac_scale_expr)
            call parse_node_write_rec (global%pn_fac_scale_expr,u)
            call msg_message ("Factorization scale expression superseding scale defintion.")
        else
            call msg_message ("No factorization scale expression defined")
        end if
    case ("renormalization_scale")
        if (associated (global%pn_ren_scale_expr)) then
            call parse_node_write_rec (global%pn_ren_scale_expr)
            call parse_node_write_rec (global%pn_ren_scale_expr,u)
            call msg_message ("Renormalization scale expression superseding scale defintion.")
        else
            call msg_message ("No renormalization scale expression defined")
        end if
    case ("analysis")
        if (associated (global%pn_analysis_lexpr)) then
            call parse_node_write_rec (global%pn_analysis_lexpr)
            call parse_node_write_rec (global%pn_analysis_lexpr, u)
        else
            call msg_message ("No cut expression defined")
        end if
    case ("expect")
        call expect_summary ()
    case ("?")
        if (associated (global%model)) then
            call var_list_write (global%var_list, only_type=V_LOG, &
                                model_name = name)
            call var_list_write (global%var_list, only_type=V_LOG, &
                                model_name = name, unit=u)
        else
            call var_list_write (global%var_list, only_type=V_LOG)
            call var_list_write (global%var_list, only_type=V_LOG, unit=u)
        end if
    case ("intrinsic")
        if (associated (global%model)) then
            call var_list_write (global%var_list, intrinsic=.true., &
                                model_name = name)

```

```

        call var_list_write (global%var_list, intrinsic=.true., &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, intrinsic=.true.)
        call var_list_write (global%var_list, intrinsic=.true., unit=u)
    end if
case ("int")
    if (associated (global%model)) then
        call var_list_write (global%var_list, only_type=V_INT, &
                             model_name = name)
        call var_list_write (global%var_list, only_type=V_INT, &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, only_type=V_INT)
        call var_list_write (global%var_list, only_type=V_INT, unit=u)
    end if
case ("real")
    if (associated (global%model)) then
        call var_list_write (global%var_list, only_type=V_REAL, &
                             model_name = name)
        call var_list_write (global%var_list, only_type=V_REAL, &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, only_type=V_REAL)
        call var_list_write (global%var_list, only_type=V_REAL, unit=u)
    end if
case ("complex")
    if (associated (global%model)) then
        call var_list_write (global%var_list, only_type=V_CMPLX, &
                             model_name = name)
        call var_list_write (global%var_list, only_type=V_CMPLX, &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, only_type=V_CMPLX)
        call var_list_write (global%var_list, only_type=V_CMPLX, unit=u)
    end if
case ("alias")
    if (associated (global%model)) then
        call var_list_write (global%var_list, only_type=V_PDG, &
                             model_name = name)
        call var_list_write (global%var_list, only_type=V_PDG, &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, only_type=V_PDG)
        call var_list_write (global%var_list, only_type=V_PDG, unit=u)
    end if
case ("$$") !$ sign
    if (associated (global%model)) then
        call var_list_write (global%var_list, only_type=V_STR, &
                             model_name = name)
        call var_list_write (global%var_list, only_type=V_STR, &
                             model_name = name, unit=u)
    else
        call var_list_write (global%var_list, only_type=V_STR)

```

```

        call var_list_write (global%var_list, only_type=V_STR, unit=u)
    end if
case ("n_calls", "num_id", &
    "integral", "error", "accuracy", "chi2", "efficiency")
    call var_list_write (global%var_list, prefix=char(show%name(i)))
    call var_list_write (global%var_list, prefix=char(show%name(i)), &
        unit=u)
case ("<lexpr>")
    lval = eval_log (show%pn_expr(i)%ptr, global%var_list, &
        is_known=is_known)
    if (is_known) then
        call var_entry_init_log (show%value(i), &
            var_str ("logical value"), lval)
    else
        call var_entry_init_log (show%value(i), &
            var_str ("logical value"))
    end if
    call var_entry_write (show%value(i))
    call var_entry_write (show%value(i), unit=u)
case ("<expr>")
    call eval_numeric (show%pn_expr(i)%ptr, global%var_list, &
        ival=ival, rval=rval, cval=cval, &
        is_known=is_known, result_type=result_type)
    select case (result_type)
    case (V_INT)
        if (is_known) then
            call var_entry_init_int (show%value(i), &
                var_str ("integer value"), ival)
        else
            call var_entry_init_int (show%value(i), &
                var_str ("integer value"))
        end if
    case (V_REAL)
        if (is_known) then
            call var_entry_init_real (show%value(i), &
                var_str ("real value"), rval)
        else
            call var_entry_init_real (show%value(i), &
                var_str ("real value"))
        end if
    case (V_CMPLX)
        if (is_known) then
            call var_entry_init_cmplx (show%value(i), &
                var_str ("complex value"), cval)
        else
            call var_entry_init_cmplx (show%value(i), &
                var_str ("complex value"))
        end if
    end select
    call var_entry_write (show%value(i))
    call var_entry_write (show%value(i), unit=u)
case ("<cexpr>")
    aval = eval_pdg_array (show%pn_expr(i)%ptr, global%var_list, &
        is_known=is_known)

```

```

        if (is_known) then
            call var_entry_init_pdg_array (show%value(i), &
                var_str ("alias value"), aval)
        else
            call var_entry_init_pdg_array (show%value(i), &
                var_str ("alias value"))
        end if
        call var_entry_write (show%value(i))
        call var_entry_write (show%value(i), unit=u)
    case ("<sexpr>")
        sval = eval_string (show%pn_expr(i)%ptr, global%var_list, &
            is_known=is_known)
        if (is_known) then
            call var_entry_init_string (show%value(i), &
                var_str ("string value"), sval)
        else
            call var_entry_init_string (show%value(i), &
                var_str ("string value"))
        end if
        call var_entry_write (show%value(i))
        call var_entry_write (show%value(i), unit=u)
    case default
        select case (char (extract (show%name(i), 1, 2)))
        case ("L ")
            name = extract (show%name(i), 3)
            prc_lib => process_library_store_get_ptr (name)
            if (associated (prc_lib)) then
                call process_library_write (prc_lib)
                call process_library_write (prc_lib, unit=u)
            else
                call msg_message ("Library '" // char (name) &
                    // "' not loaded")
            end if
        case ("A ")
            name = extract (show%name(i), 3)
            call var_list_write_var (global%var_list, name, &
                model_name = name, type = V_PDG)
            call var_list_write_var (global%var_list, name, &
                model_name = name, type = V_PDG, unit=u)
        case default
            if (associated (global%model)) then
                call var_list_write_var (global%var_list, show%name(i), &
                    model_name = name)
                call var_list_write_var (global%var_list, show%name(i), &
                    model_name = name, unit=u)
            else
                call var_list_write_var (global%var_list, show%name(i))
                call var_list_write_var (global%var_list, show%name(i), &
                    unit=u)
            end if
        end select
    end select
end do
end if

```

```

flush (u)
end subroutine cmd_show_execute

```

### Compare values of variables to expectation

The implementation is similar to the `show` command. There are just two arguments: two values that should be compared. For providing local values for the numerical tolerance, the command has a local argument list.

If the expectation fails, an error condition is recorded.

```

<CCC Commands: types>+≡
type :: cmd_expect_t
  private
  type(parse_node_t), pointer :: pn_lexpr => null ()
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_expect_t

```

Finalize.

```

<CCC Commands: procedures>+≡
subroutine cmd_expect_final (expect)
  type(cmd_expect_t), intent(inout) :: expect
  if (associated (expect%options)) then
    call command_list_final (expect%options)
    deallocate (expect%options)
  end if
end subroutine cmd_expect_final

```

Compile. This merely assigns the parse node, the actual compilation is done at execution. This is necessary because the origin of variables (local/global) may change during execution.

```

<CCC Commands: procedures>+≡
subroutine cmd_expect_compile (expect, pn, global)
  type(cmd_expect_t), pointer :: expect
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_opt
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  pn_opt => parse_node_get_next_ptr (pn_arg)
  allocate (expect)
  call rt_data_local_init (expect%local, global)
  if (associated (pn_opt)) then
    allocate (expect%options)
    call command_list_compile (expect%options, pn_opt, expect%local)
  end if
  expect%pn_lexpr => parse_node_get_sub_ptr (pn_arg)
  call rt_data_local_reset (expect%local)
end subroutine cmd_expect_compile

```

Execute. Evaluate both arguments, print them and their difference (if numerical), and whether they agree. Record the result.

```

<CCC Commands: procedures>+≡

```



```

subroutine cmd_expect_execute (expect, global)
  type(cmd_expect_t), intent(inout) :: expect
  type(rt_data_t), intent(inout), target :: global
  logical :: success, is_known
  integer :: u
  u = logfile_unit ()
  call rt_data_link (expect%local, global)
  if (associated (expect%options)) then
    call command_list_execute (expect%options, expect%local)
  end if
  success = &
    eval_log (expect%pn_lexpr, expect%local%var_list, is_known=is_known)
  if (is_known) then
    if (success) then
      call msg_message ("expect: success")
    else
      call msg_error ("expect: failure")
    end if
  else
    call msg_error ("expect: undefined result")
    success = .false.
  end if
  call expect_record (success)
  call rt_data_restore (global, expect%local)
end subroutine cmd_expect_execute

```

## Beams

The beam command includes both beam and structure-function definition.

*<Commands: types>+≡*

```

type, extends (command_t) :: cmd_beams_t
  private
  integer :: n_in = 0
  type(parse_node_p), dimension(:), allocatable :: pn_pdg
  integer :: n_sf_record = 0
  integer, dimension(:), allocatable :: n_entry
  type(parse_node_p), dimension(:,:), allocatable :: pn_sf_entry
!   type(command_list_t), pointer :: options => null ()
  type(rt_data_t), pointer :: local
!   type(strfun_pair_t), dimension(:), allocatable :: strfun_pair
contains
  <Commands: cmd beams: TBP>
end type cmd_beams_t

```

*<CCC Commands: types>+≡*

```

type :: strfun_def_t
  private
  integer :: type = STRF_NONE
  type(command_list_t), pointer :: options => null ()
  type(parse_node_t), pointer :: pn_user_name => null ()
  type(rt_data_t) :: local
end type strfun_def_t

```

```

type :: strfun_pair_t
  private
  integer :: n
  type(strfun_def_t), dimension(2) :: def
end type strfun_pair_t

```

Delete the eval trees.

```

<CCC Commands: procedures>+≡
  subroutine cmd_beams_final (beams)
    type(cmd_beams_t), intent(inout) :: beams
    if (allocated (beams%pn_pdg)) then
      deallocate (beams%pn_pdg)
    end if
    if (associated (beams%options)) then
      call command_list_final (beams%options)
      deallocate (beams%options)
    end if
  end subroutine cmd_beams_final

```

Output. The particle expressions are not resolved.

```

<Commands: cmd beams: TBP>≡
  procedure :: write => cmd_beams_write

<Commands: procedures>+≡
  subroutine cmd_beams_write (cmd, unit, indent)
    class(cmd_beams_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    select case (cmd%n_in)
    case (1)
      write (u, "(1x,A)") "beams: 1 [decay]"
    case (2)
      write (u, "(1x,A)") "beams: 2 [scattering]"
    case default
      write (u, "(1x,A)") "beams: [undefined]"
    end select
    if (allocated (cmd%n_entry)) then
      if (cmd%n_sf_record > 0) then
        write (u, "(1x,A,99(1x,I0))") "structure function entries:", &
          cmd%n_entry
      end if
    end if
  end subroutine cmd_beams_write

```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

```

<Commands: cmd beams: TBP>+≡
  procedure :: compile => cmd_beams_compile

```

*<Commands: procedures>+≡*

```

subroutine cmd_beams_compile (cmd, global)
  class(cmd_beams_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_beam_def, pn_beam_spec
  type(parse_node_t), pointer :: pn_beam_list, pn_opt
  type(parse_node_t), pointer :: pn_codes
  type(parse_node_t), pointer :: pn_strfun_seq, pn_strfun_pair
  type(parse_node_t), pointer :: pn_strfun_def
  integer :: i
  pn_beam_def => parse_node_get_sub_ptr (cmd%pn, 3)
  pn_beam_spec => parse_node_get_sub_ptr (pn_beam_def)
  pn_strfun_seq => parse_node_get_next_ptr (pn_beam_spec)
  pn_beam_list => parse_node_get_sub_ptr (pn_beam_spec)
!   pn_opt => parse_node_get_next_ptr (pn_beam_list)
  cmd%local => global
  cmd%n_in = parse_node_get_n_sub (pn_beam_list)
  allocate (cmd%pn_pdg (cmd%n_in))
  pn_codes => parse_node_get_sub_ptr (pn_beam_list)
  do i = 1, cmd%n_in
    cmd%pn_pdg(i)%ptr => pn_codes
    pn_codes => parse_node_get_next_ptr (pn_codes)
  end do
  if (associated (pn_strfun_seq)) then
    cmd%n_sf_record = parse_node_get_n_sub (pn_beam_def) - 1
    allocate (cmd%n_entry (cmd%n_sf_record), source = 1)
    allocate (cmd%pn_sf_entry (2, cmd%n_sf_record))
    do i = 1, cmd%n_sf_record
      pn_strfun_pair => parse_node_get_sub_ptr (pn_strfun_seq, 2)
      pn_strfun_def => parse_node_get_sub_ptr (pn_strfun_pair)
      cmd%pn_sf_entry(1,i)%ptr => pn_strfun_def
      pn_strfun_def => parse_node_get_next_ptr (pn_strfun_def)
      cmd%pn_sf_entry(2,i)%ptr => pn_strfun_def
      if (associated (pn_strfun_def)) cmd%n_entry(i) = 2
      pn_strfun_seq => parse_node_get_next_ptr (pn_strfun_seq)
    end do
  else
    allocate (cmd%n_entry (0))
    allocate (cmd%pn_sf_entry (0, 0))
  end if
!   call rt_data_local_reset (cmd%local)
end subroutine cmd_beams_compile

```

*<CCC Commands: procedures>+≡*

```

subroutine strfun_pair_compile (strfun_pair, pn_strfun_pair, global)
  type(strfun_pair_t), intent(out) :: strfun_pair
  type(parse_node_t), intent(in), target :: pn_strfun_pair
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_strfun_def
  integer :: i
  strfun_pair%n = parse_node_get_n_sub (pn_strfun_pair)
  pn_strfun_def => parse_node_get_sub_ptr (pn_strfun_pair)
  do i = 1, strfun_pair%n
    call strfun_def_compile (strfun_pair%def(i), pn_strfun_def, global)
  end do

```

```

        pn_strfun_def => parse_node_get_next_ptr (pn_strfun_def)
    end do
end subroutine strfun_pair_compile

subroutine strfun_def_compile (strfun_def, pn_strfun_def, global)
    type(strfun_def_t), intent(out) :: strfun_def
    type(parse_node_t), intent(in), target :: pn_strfun_def
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_key, pn_opt, pn_arg
    pn_key => parse_node_get_sub_ptr (pn_strfun_def)
    pn_opt => parse_node_get_next_ptr (pn_key)
    select case (char (parse_node_get_rule_key (pn_key)))
    case ("none")
        strfun_def%type = STRF_NONE
    case ("lhpdf")
        strfun_def%type = STRF_LHAPDF
    case ("pdf_builtin")
        strfun_def%type = STRF_PDF_BUILTIN
    case ("isr")
        strfun_def%type = STRF_ISR
    case ("epa")
        strfun_def%type = STRF_EPA
    case ("ewa")
        strfun_def%type = STRF_EWA
    case ("circe1")
        strfun_def%type = STRF_CIRCE1
    case ("circe2")
        strfun_def%type = STRF_CIRCE2
    case ("energy_scan")
        strfun_def%type = STRF_ESCAN
    case ("beam_events")
        strfun_def%type = STRF_BEVT
    case ("user_sf_spec")
        strfun_def%type = STRF_USER
        pn_arg => parse_node_get_sub_ptr (pn_key, 2)
        strfun_def%pn_user_name => parse_node_get_sub_ptr (pn_arg)
    end select
    call rt_data_local_init (strfun_def%local, global)
    if (associated (pn_opt)) then
        allocate (strfun_def%options)
        call command_list_compile &
            (strfun_def%options, pn_opt, strfun_def%local)
    end if
    call rt_data_local_reset (strfun_def%local)
end subroutine strfun_def_compile

```

Command execution: Beam initialization. The output is a beam-data object, which is copied to the global data block.

Structure functions are configured in the option section. They are linked to the global data block, where they can be found by the integration command.

Note: polarization, beam structure, local options not implemented yet.

*(Commands: cmd beams: TBP)+≡*

```

procedure :: execute => cmd_beams_execute

```

{Commands: procedures}+≡

```

subroutine cmd_beams_execute (cmd, global)
  class(cmd_beams_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(pdg_array_t) :: pdg_array
  integer, dimension(:), allocatable :: pdg
!   logical, dimension(2) :: p_known
!   real(default), dimension(2) :: p
  logical :: sqrts_known ! , alpha_known, theta_known, phi_known
  real(default) :: sqrts ! , alpha, theta, phi
!   real(default) :: beams_theta, beams_phi
  type(flavor_t), dimension(:), allocatable :: flv
  type(parse_node_t), pointer :: pn_key
  type(string_t) :: sf_name
!   type(beam_polarization_t), dimension(2) :: bp
!   type(polarization_t), dimension(2) :: pol
  integer :: i, j, u
!   logical :: polarized
!   u = logfile_unit ()
  call lhpdf_status_reset (global%lhpdf_status)
  call pdf_builtin_status_reset (global%pdf_builtin_status)
!   call rt_data_link (cmd%local, global)
!   if (associated (cmd%options)) then
!     call command_list_execute (cmd%options, cmd%local)
!   end if
!   polarized = .true.
  allocate (flv (cmd%n_in))
  do i = 1, cmd%n_in
    pdg_array = eval_pdg_array (cmd%pn_pdg(i)%ptr, cmd%local%var_list)
    pdg = pdg_array
    select case (size (pdg))
    case (1)
      call flavor_init (flv(i), pdg(1), cmd%local%model)
    case default
      call msg_fatal ("Beams: beam particles must be unique")
    end select
  end do
!   if (polarized .and. associated (cmd%local%beam_polarization)) then
!     if (size (cmd%local%beam_polarization) == cmd%n_in) then
!       bp(1:cmd%n_in) = cmd%local%beam_polarization
!     else
!       call msg_error ("the number of incoming particles differs " &
!         // "between beam and polarization setup - ignoring polarization")
!       polarized = .false.
!     end if
!   else
!     polarized = .false.
!     select case (cmd%n_in)
!     case (1)
!       call beam_polarization_init_trivial (bp(i))
!     case (2)
!       call beam_polarization_init_none (bp(i))
!     end select
!   end if

```

```

!         pol(i) = beam_polarization2polarization (bp(i), flv(i), &
!             decay=(size (bp) == 1))
!     end do
!     sqrts_known = var_list_is_known (cmd%local%var_list, "sqrts")
!     p_known(1) = var_list_is_known (cmd%local%var_list, "beam1_momentum")
!     p_known(2) = var_list_is_known (cmd%local%var_list, "beam2_momentum")
!     alpha_known = var_list_is_known (cmd%local%var_list, "crossing_angle")
!     theta_known = var_list_is_known (cmd%local%var_list, "beams_theta")
!     phi_known = var_list_is_known (cmd%local%var_list, "beams_phi")
!     sqrts = var_list_get_rval (cmd%local%var_list, "sqrts")
!     p(1) = var_list_get_rval (cmd%local%var_list, "beam1_momentum")
!     p(2) = var_list_get_rval (cmd%local%var_list, "beam2_momentum")
!     alpha = var_list_get_rval (cmd%local%var_list, "crossing_angle")
!     theta = var_list_get_rval (cmd%local%var_list, "beams_theta")
!     phi = var_list_get_rval (cmd%local%var_list, "beams_phi")
!     select case (cmd%n_in)
!     case (2)
!         if (sqrts_known) then
!             if (any (p_known)) call msg_error &
!                 ("Beam setup: sqrts and beam momenta must not be set " &
!                  // "simultaneously; using sqrts only")
!             if (alpha_known) call msg_fatal &
!                 ("Beam setup: sqrts and crossing angle must not be set " &
!                  // "simultaneously (set beam1_momentum and beam2_momentum " &
!                  // "instead)")
!             else
!                 if (.not. all (p_known)) call msg_fatal &
!                     ("Beam setup: either sqrts or both beam momenta must be set.")
!             end if
!         end select
!     select case (cmd%n_in)
!     case (1)
!         if (p_known(1) .or. theta_known .or. phi_known) then
!             call beam_data_init_decay &
!                 (global%beam_data, flv, pol(1:1), p(1), theta, phi)
!         else
!             call beam_data_init_decay (global%beam_data, flv, pol(1:1))
!         end if
!         call msg_bug ("Beams: decay setup not implemented yet")
!     case (2)
!         if (sqrts_known) then
!             if (theta_known .or. phi_known) then
!                 call beam_data_init_sqrts (global%beam_data, &
!                     sqrts, flv, pol, 0._default, theta=theta, phi=phi)
!             else
!                 call beam_data_init_sqrts (global%beam_data, &
!                     sqrts, flv) !, pol)
!             end if
!         else if (alpha_known) then
!             if (theta_known .or. phi_known) then
!                 call beam_data_init_momenta (global%beam_data, &
!                     p, flv, pol, alpha, theta, phi)
!             else
!                 call beam_data_init_momenta (global%beam_data, &

```

```

!           p, flv, pol, alpha)
!       end if
!   else
!       if (theta_known .or. phi_known) then
!           call beam_data_init_momenta (global%beam_data, &
!               p, flv, pol, theta=theta, phi=phi)
!       else
!           call beam_data_init_momenta (global%beam_data, &
!               p, flv, pol)
!       end if
!   end if

call global%beam_structure%init (cmd%n_entry, cmd%local%var_list)
do i = 1, cmd%n_sf_record
    do j = 1, cmd%n_entry(i)
        pn_key => parse_node_get_sub_ptr (cmd%pn_sf_entry(j,i)%ptr)
        sf_name = parse_node_get_key (pn_key)
        call global%beam_structure%set_entry (i, j, sf_name)
    end do
end do

end select
!   call beam_data_write (global%beam_data, verbose=.false.)
!   call beam_data_write (global%beam_data, verbose=.false., unit=u)
!   if (polarized) then
!       do i = 1, cmd%n_in
!           if (output_unit () >= 0) write (output_unit (), '(1x,A)') &
!               "polarization of '" // char (flavor_get_name (flv(i))) // "':"
!           if (u >= 0) write (u, '(1x,A)') &
!               "polarization of '" // char (flavor_get_name (flv(i))) // "':"
!           call beam_polarization_write (cmd%local%beam_polarization(i))
!           call beam_polarization_write (cmd%local%beam_polarization(i), u)
!       end do
!   end if
!   if (u >= 0) flush (u)
!   call rt_data_restore (global, cmd%local)
end subroutine cmd_beams_execute

```

*<CCC Commands: procedures>+≡*

```

!! !! subroutine strfun_pair_register (strfun_pair, global)
!! !!   type(strfun_pair_t), intent(inout) :: strfun_pair
!! !!   type(rt_data_t), intent(inout), target :: global
!! !!   logical, dimension(2) :: affects_beam
!! !!   select case (strfun_pair%n)
!! !!   case (1)
!! !!       affects_beam = .true.
!! !!       call strfun_def_register (strfun_pair%def(1), affects_beam, global)
!! !!   case (2)
!! !!       affects_beam = (/ .true., .false. /)
!! !!       call strfun_def_register (strfun_pair%def(1), affects_beam, global)
!! !!       affects_beam = (/ .false., .true. /)
!! !!       call strfun_def_register (strfun_pair%def(2), affects_beam, global)
!! !!   end select
!! !! end subroutine strfun_pair_register

```

```

!! !!
!! !! subroutine strfun_def_register (strfun_def, affects_beam, global)
!! !!   type(strfun_def_t), intent(inout) :: strfun_def
!! !!   logical, dimension(2), intent(in) :: affects_beam
!! !!   type(rt_data_t), intent(inout), target :: global
!! !!   call rt_data_link (strfun_def%local, global)
!! !!   if (associated (strfun_def%options)) then
!! !!     call command_list_execute (strfun_def%options, strfun_def%local)
!! !!   end if
!! !!   select case (strfun_def%type)
!! !!   case (STRF_LHAPDF)
!! !!     call sf_list_register_lhapdf &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%lhpdf_status, global%model, global%beam_data%flv, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_PDF_BUILTIN)
!! !!     call sf_list_register_pdf_builtin &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%pdf_builtin_status, global%model, global%beam_data%flv, &
!! !!         strfun_def%local%os_data%pdf_builtin_datapath, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_ISR)
!! !!     call sf_list_register_isr &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%model, global%beam_data%flv, global%beam_data%sqrts, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_EPA)
!! !!     call sf_list_register_epa &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%model, global%beam_data%flv, global%beam_data%sqrts, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_EWA)
!! !!     call msg_warning ("EWA structure function not yet fully implemented")
!! !!     call sf_list_register_ewa &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%model, global%beam_data%flv, global%beam_data%sqrts, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_CIRCE1)
!! !!     call sf_list_register_circe1 &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%model, global%beam_data%flv, global%beam_data%sqrts, &
!! !!         global%rng, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_CIRCE2)
!! !!     call sf_list_register_circe2 &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%beam_data%flv, global%beam_data%sqrts, &
!! !!         global%rng, global%os_data%whizard_circe2path, &
!! !!         strfun_def%local%var_list)
!! !!   case (STRF_ESCAN)
!! !!     call sf_list_register_escan &
!! !!       (global%sf_list, affects_beam, &
!! !!         global%beam_data%flv, global%beam_data%sqrts, &
!! !!         strfun_def%local%var_list)

```



```

!! !! case (STRF_BEVT)
!! !!   call sf_list_register_beam_events &
!! !!     (global%sf_list, affects_beam, &
!! !!       global%beam_data%flv, global%os_data%whizard_beamsimpath, &
!! !!       strfun_def%local%var_list)
!! !! case (STRF_USER)
!! !!   call sf_list_register_user &
!! !!     (global%sf_list, affects_beam, &
!! !!       global%model, global%beam_data%flv, &
!! !!       eval_string (strfun_def%pn_user_name, strfun_def%local%var_list), &
!! !!       strfun_def%local%var_list)
!! !!   end select
!! !!   call rt_data_restore (global, strfun_def%local)
!! !! end subroutine strfun_def_register

```

The individual register procedures. They unpack data from the local var list, allocate structure function data blocks and fill them. Depending on the structure function and on the beams on which it should apply, one or two such entries are added to the structure function list. For double structure functions, it is useful to add a pair-mapping.

*(CCC Commands: procedures)+≡*

```

!! !! subroutine sf_list_register_lhapdf (sf_list, affects_beam, &
!! !!   lhpdf_status, model, flv, var_list)
!! !!   type(sf_list_t), intent(inout) :: sf_list
!! !!   logical, dimension(2), intent(in) :: affects_beam
!! !!   type(lhapdf_status_t), intent(inout) :: lhpdf_status
!! !!   type(model_t), intent(in), target :: model
!! !!   type(flavor_t), dimension(2), intent(in) :: flv
!! !!   type(var_list_t), intent(in) :: var_list
!! !!   type(string_t) :: lhpdf_file, lhpdf_dir
!! !!   integer :: lhpdf_member, lhpdf_photon_scheme
!! !!   type(sf_data_t), pointer :: sf_data
!! !!   integer :: i
!! !!   lhpdf_dir = &
!! !!     var_list_get_sval (var_list, var_str ("lhpdf_dir")) ! $
!! !!   lhpdf_file = &
!! !!     var_list_get_sval (var_list, var_str ("lhpdf_file")) ! $
!! !!   lhpdf_member = &
!! !!     var_list_get_ival (var_list, var_str ("lhpdf_member"))
!! !!   lhpdf_photon_scheme = &
!! !!     var_list_get_ival (var_list, var_str ("lhpdf_photon_scheme"))
!! !!   do i = 1, 2
!! !!     if (affects_beam(i)) then
!! !!       allocate (sf_data)
!! !!       call sf_data_init_lhapdf (sf_data, i, lhpdf_status, &
!! !!         model, flv(i), &
!! !!         lhpdf_dir, lhpdf_file, lhpdf_member, lhpdf_photon_scheme)
!! !!       call sf_list_append (sf_list, sf_data)
!! !!     end if
!! !!   end do
!! !!   if (all (affects_beam)) then
!! !!     call sf_data_setup_mapping &
!! !!       (sf_data, SFM_PAIR, (/ 0, 1 /), 2._default)

```

```
!! !! end if
!! !! end subroutine sf_list_register_lhapdf
```

*{CCC Commands: procedures}+≡*

```
!! subroutine sf_list_register_pdf_builtin (sf_list, affects_beam, &
!!     pdf_status, model, flv, datapath, var_list)
!!     type(sf_list_t), intent(inout) :: sf_list
!!     logical, dimension(2), intent(in) :: affects_beam
!!     type(pdf_builtin_status_t), intent(inout) :: pdf_status
!!     type(model_t), intent(in), target :: model
!!     type(flavor_t), dimension(2), intent(in) :: flv
!!     type(string_t), intent(in) :: datapath
!!     type(var_list_t), intent(in) :: var_list
!!     logical :: pdf_builtin_have_name
!!     type(string_t) :: pdf_builtin_prefix, pdf_builtin_name
!!     type(sf_data_t), pointer :: sf_data
!!     integer :: i
!!     pdf_builtin_have_name = &
!!         var_list_is_known (var_list, var_str ("pdf_builtin_set"))
!!     if (pdf_builtin_have_name) &
!!         pdf_builtin_name = &
!!             var_list_get_sval (var_list, var_str ("pdf_builtin_set"))
!!     pdf_builtin_have_name = trim (pdf_builtin_name) /= ""
!!     pdf_builtin_prefix = ""
!!     if (var_list_is_known (var_list, var_str ("pdf_builtin_path"))) &
!!         pdf_builtin_prefix = &
!!             var_list_get_sval (var_list, var_str ("pdf_builtin_path"))
!!     if (trim (pdf_builtin_prefix) == "") pdf_builtin_prefix = datapath
!!     do i = 1, 2
!!         if (affects_beam(i)) then
!!             allocate (sf_data)
!!             if (pdf_builtin_have_name) then
!!                 call sf_data_init_pdf_builtin (sf_data, i, &
!!                     pdf_status, model, flv(i), &
!!                     name=pdf_builtin_name, path=pdf_builtin_prefix)
!!             else
!!                 call sf_data_init_pdf_builtin (sf_data, i, &
!!                     pdf_status, model, flv(i), &
!!                     path=pdf_builtin_prefix)
!!             end if
!!             call sf_list_append (sf_list, sf_data)
!!         end if
!!     end do
!!     if (all (affects_beam)) then
!!         call sf_data_setup_mapping &
!!             (sf_data, SFM_PAIR, (/ 0, 1 /), 2._default)
!!     end if
!! end subroutine sf_list_register_pdf_builtin
```

*{CCC Commands: procedures}+≡*

```
!! subroutine sf_list_register_isr (sf_list, affects_beam, &
!!     model, flv, sqrts, var_list)
!!     type(sf_list_t), intent(inout) :: sf_list
```

```

!! logical, dimension(2), intent(in) :: affects_beam
!! type(model_t), intent(in), target :: model
!! type(flavor_t), dimension(2), intent(in) :: flv
!! real(default), intent(in) :: sqrts
!! type(var_list_t), intent(in) :: var_list
!! real(default) :: isr_alpha, isr_q_max, isr_mass
!! integer :: isr_order
!! logical :: isr_recoil
!! type(sf_data_t), pointer :: sf_data
!! integer :: i
!! isr_alpha = var_list_get_rval (var_list, var_str ("isr_alpha"))
!! if (isr_alpha == 0) then
!!     isr_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
!!                 ** 2 / (4 * pi)
!! end if
!! isr_q_max = var_list_get_rval (var_list, var_str ("isr_q_max"))
!! if (isr_q_max == 0) then
!!     isr_q_max = sqrts
!! end if
!! isr_mass = var_list_get_rval (var_list, var_str ("isr_mass"))
!! isr_order = var_list_get_ival (var_list, var_str ("isr_order"))
!! isr_recoil = var_list_get_lval (var_list, var_str ("?isr_recoil"))
!! do i = 1, 2
!!     if (affects_beam(i)) then
!!         allocate (sf_data)
!!         if (isr_mass /= 0) then
!!             call sf_data_init_isr (sf_data, i, &
!!                                   model, flv(i), isr_recoil, isr_alpha, isr_q_max, isr_mass, &
!!                                   order=isr_order)
!!         else
!!             call sf_data_init_isr (sf_data, i, &
!!                                   model, flv(i), isr_recoil, isr_alpha, isr_q_max, &
!!                                   order=isr_order)
!!         end if
!!         call sf_list_append (sf_list, sf_data)
!!     end if
!! end do
!! ! No pair mapping
!! end subroutine sf_list_register_isr

```

*<CCC Commands: procedures>+≡*

```

!! !! subroutine sf_list_register_epa (sf_list, affects_beam, &
!! !!     model, flv, sqrts, var_list)
!! !! type(sf_list_t), intent(inout) :: sf_list
!! !! logical, dimension(2), intent(in) :: affects_beam
!! !! type(model_t), intent(in), target :: model
!! !! type(flavor_t), dimension(2), intent(in) :: flv
!! !! real(default), intent(in) :: sqrts
!! !! type(var_list_t), intent(in) :: var_list
!! !! real(default) :: epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass
!! !! logical :: epa_recoil
!! !! type(sf_data_t), pointer :: sf_data
!! !! integer :: i
!! !! epa_alpha = var_list_get_rval (var_list, var_str ("epa_alpha"))

```

```

!! !! if (epa_alpha == 0) then
!! !!     epa_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
!! !!         ** 2 / (4 * pi)
!! !! end if
!! !! epa_x_min = var_list_get_rval (var_list, var_str ("epa_x_min"))
!! !! epa_q_min = var_list_get_rval (var_list, var_str ("epa_q_min"))
!! !! epa_e_max = var_list_get_rval (var_list, var_str ("epa_e_max"))
!! !! if (epa_e_max == 0) then
!! !!     epa_e_max = sqrts
!! !! end if
!! !! epa_mass = var_list_get_rval (var_list, var_str ("epa_mass"))
!! !! epa_recoil = var_list_get_lval (var_list, var_str ("?epa_recoil"))
!! !! do i = 1, 2
!! !!     if (affects_beam(i)) then
!! !!         allocate (sf_data)
!! !!         if (epa_mass /= 0) then
!! !!             call sf_data_init_epa (sf_data, i, &
!! !!                 model, flv(i), epa_recoil, &
!! !!                 epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass)
!! !!         else
!! !!             call sf_data_init_epa (sf_data, i, &
!! !!                 model, flv(i), epa_recoil, &
!! !!                 epa_alpha, epa_x_min, epa_q_min, epa_e_max)
!! !!         end if
!! !!         call sf_list_append (sf_list, sf_data)
!! !!     end if
!! !! end do
!! !! if (all (affects_beam)) then
!! !!     if (epa_recoil) then
!! !!         call sf_data_setup_mapping &
!! !!             (sf_data, SFM_PAIR, (/ -2, 1 /), 1._default)
!! !!     else
!! !!         call sf_data_setup_mapping &
!! !!             (sf_data, SFM_PAIR, (/ 0, 1 /), 1._default)
!! !!     end if
!! !! end if
!! !! end subroutine sf_list_register_epa

```

*<CCC Commands: procedures>+≡*

```

!! !! subroutine sf_list_register_ewa (sf_list, affects_beam, &
!! !!     model, flv, sqrts, var_list)
!! !! type(sf_list_t), intent(inout) :: sf_list
!! !! logical, dimension(2), intent(in) :: affects_beam
!! !! type(model_t), intent(in), target :: model
!! !! type(flavor_t), dimension(2), intent(in) :: flv
!! !! real(default), intent(in) :: sqrts
!! !! type(var_list_t), intent(in) :: var_list
!! !! real(default) :: ewa_x_min, ewa_q_min, ewa_pt_max, ewa_mass, ewa_sqrts
!! !! logical :: ewa_keep_momentum, ewa_keep_energy
!! !! type(sf_data_t), pointer :: sf_data
!! !! integer :: i
!! !! do i = 1, 2
!! !!     if (affects_beam(i)) then
!! !!         allocate (sf_data)

```

```

!! !!      ewa_x_min = var_list_get_rval (var_list, var_str ("ewa_x_min"))
!! !!      ewa_q_min = var_list_get_rval (var_list, var_str ("ewa_q_min"))
!! !!      ewa_pt_max = var_list_get_rval (var_list, var_str ("ewa_pt_max"))
!! !!      if (ewa_pt_max == 0) then
!! !!          ewa_pt_max = sqrts
!! !!      end if
!! !!      ewa_mass = var_list_get_rval (var_list, var_str ("ewa_mass"))
!! !!      ewa_sqrts = sqrts
!! !!      ewa_keep_momentum = var_list_get_lval (var_list, &
!! !!          var_str ("?ewa_keep_momentum"))
!! !!      ewa_keep_energy = var_list_get_lval (var_list, &
!! !!          var_str ("?ewa_keep_energy"))
!! !!      if (ewa_keep_momentum .and. ewa_keep_energy) &
!! !!          call msg_fatal (" EWA cannot violate both energy " &
!! !!              // "and momentum conservation.")
!! !!      if (ewa_mass /= 0) then
!! !!          call sf_data_init_ewa (sf_data, i, &
!! !!              model, flv(i), &
!! !!              ewa_x_min, ewa_q_min, ewa_pt_max, ewa_sqrts, &
!! !!              ewa_keep_momentum, ewa_keep_energy, ewa_mass)
!! !!      else
!! !!          call sf_data_init_ewa (sf_data, i, &
!! !!              model, flv(i), &
!! !!              ewa_x_min, ewa_q_min, ewa_pt_max, ewa_sqrts, &
!! !!              ewa_keep_momentum, ewa_keep_energy)
!! !!      end if
!! !!      call sf_list_append (sf_list, sf_data)
!! !!  end if
!! !!  end do
!! !!  if (all (affects_beam)) then
!! !!      call sf_data_setup_mapping &
!! !!          (sf_data, SFM_PAIR, (/ 0, 1 /), 1._default)
!! !!  end if
!! !!  end subroutine sf_list_register_ewa

```

{CCC Commands: procedures}+≡

```

subroutine sf_list_register_circe1 (sf_list, affects_beam, &
    model, flv, sqrts, rng, var_list)
    type(sf_list_t), intent(inout) :: sf_list
    logical, dimension(2), intent(in) :: affects_beam
    type(model_t), intent(in), target :: model
    type(flavor_t), dimension(2), intent(in) :: flv
    real(default), intent(in) :: sqrts
    type(tao_random_state), intent(in), target :: rng
    type(var_list_t), intent(in) :: var_list
    real(default) :: circe1_sqrts
    logical, dimension(2) :: circe1_photon
    logical :: circe1_generate, circe1_map
    integer :: circe1_ver, circe1_rev, circe1_acc, circe1_chat
    type(sf_data_t), pointer :: sf_data
    if (all (affects_beam)) then
        allocate (sf_data)
        if (var_list_is_known (var_list, var_str ("circe1_sqrts"))) then
            circe1_sqrts = var_list_get_rval (var_list, var_str ("circe1_sqrts"))

```

```

else
    circe1_sqrts = sqrts
end if
circe1_photon(1) = &
    var_list_get_lval (var_list, var_str ("?circe1_photon1"))
circe1_photon(2) = &
    var_list_get_lval (var_list, var_str ("?circe1_photon2"))
circe1_generate = &
    var_list_get_lval (var_list, var_str ("?circe1_generate"))
circe1_map = &
    var_list_get_lval (var_list, var_str ("?circe1_map"))
circe1_ver = &
    var_list_get_ival (var_list, var_str ("circe1_ver"))
circe1_rev = &
    var_list_get_ival (var_list, var_str ("circe1_rev"))
circe1_acc = &
    var_list_get_ival (var_list, var_str ("circe1_acc"))
circe1_chat = &
    var_list_get_ival (var_list, var_str ("circe1_chat"))
call sf_data_init_circe1 (sf_data, &
    model, flv, circe1_sqrts, circe1_photon, &
    circe1_generate, rng, circe1_map, &
    circe1_ver, circe1_rev, circe1_acc, circe1_chat)
call sf_list_append (sf_list, sf_data)
else
    call msg_fatal ("CIRCE1 beamstrahlung spectrum must apply to both beams")
end if
! No pair mapping
end subroutine sf_list_register_circe1

```

*{CCC Commands: procedures}+≡*

```

subroutine sf_list_register_circe2 (sf_list, affects_beam, &
    flv, sqrts, rng, path, var_list)
type(sf_list_t), intent(inout) :: sf_list
logical, dimension(2), intent(in) :: affects_beam
type(flavor_t), dimension(2), intent(in) :: flv
real(default), intent(in) :: sqrts
type(tao_random_state), intent(in), target :: rng
type(string_t), intent(in) :: path
type(var_list_t), intent(in) :: var_list
real(default) :: circe2_sqrts
logical :: circe2_generate, circe2_map, circe2_polarized
type(string_t) :: circe2_file, circe2_design
type(sf_data_t), pointer :: sf_data
if (all (affects_beam)) then
    allocate (sf_data)
    if (var_list_is_known (var_list, var_str ("circe2_sqrts"))) then
        circe2_sqrts = var_list_get_rval (var_list, var_str ("circe2_sqrts"))
    else
        circe2_sqrts = sqrts
    end if
    circe2_generate = &
        var_list_get_lval (var_list, var_str ("?circe2_generate"))
    circe2_map = &

```

```

        var_list_get_lval (var_list, var_str ("?circe2_map"))
circe2_polarized = &
        var_list_get_lval (var_list, var_str ("?circe2_polarized"))
circe2_file = &
        var_list_get_sval (var_list, var_str ("$circe2_file")) ! $
if (circe2_file == "") call msg_fatal &
    ("CIRCE2: Data file $circe2_file must be specified") ! $
circe2_file = path // "/" // circe2_file
circe2_design = &
        var_list_get_sval (var_list, var_str ("$circe2_design")) ! $
call sf_data_init_circe2 (sf_data, &
    flv, circe2_generate, rng, &
    circe2_map, circe2_file, circe2_design, circe2_sqrts, &
    circe2_polarized)
call sf_list_append (sf_list, sf_data)
else
    call msg_fatal ("CIRCE2 spectrum must apply to both beams")
end if
! No pair mapping
end subroutine sf_list_register_circe2

```

*<CCC Commands: procedures>+≡*

```

subroutine sf_list_register_escan (sf_list, affects_beam, &
    flv, sqrts, var_list)
type(sf_list_t), intent(inout) :: sf_list
logical, dimension(2), intent(in) :: affects_beam
type(flavor_t), dimension(2), intent(in) :: flv
real(default), intent(in) :: sqrts
type(var_list_t), intent(in) :: var_list
real(default) :: escan_sqrts
type(sf_data_t), pointer :: sf_data
escan_sqrts = sqrts
allocate (sf_data)
call sf_data_init_escan (sf_data, affects_beam, flv, escan_sqrts)
call sf_list_append (sf_list, sf_data)
! No pair mapping
end subroutine sf_list_register_escan

```

*<CCC Commands: procedures>+≡*

```

subroutine sf_list_register_beam_events (sf_list, affects_beam, &
    flv, path, var_list)
type(sf_list_t), intent(inout) :: sf_list
logical, dimension(2), intent(in) :: affects_beam
type(flavor_t), dimension(2), intent(in) :: flv
type(string_t), intent(in) :: path
type(var_list_t), intent(in) :: var_list
type(sf_data_t), pointer :: sf_data
type(string_t) :: beam_events_file
logical :: beam_events_warn_eof
logical :: exist
if (all (affects_beam)) then
    allocate (sf_data)
    beam_events_file = &

```

```

        var_list_get_sval (var_list, var_str ("beam_events_file")) ! $
beam_events_warn_eof = &
        var_list_get_lval (var_list, var_str ("?beam_events_warn_eof"))
inquire (file = char (beam_events_file), exist = exist)
if (.not. exist) then
    beam_events_file = path // "/" // beam_events_file
    inquire (file = char (beam_events_file), exist = exist)
    if (.not. exist) then
        call msg_fatal ("Beam simulation data file '" &
            // char (beam_events_file) // "' not found.")
    end if
end if
call sf_data_init_beam_events &
    (sf_data, affects_beam, flv, beam_events_file, beam_events_warn_eof)
call sf_list_append (sf_list, sf_data)
else
    call msg_fatal ("Beam events simulation must apply to both beams")
end if
! No pair mapping
end subroutine sf_list_register_beam_events

```

For user structure functions, it is not evident whether they apply to single beams or to the beam pair, before the data set has been initialized. Therefore, we have to shortcut the scan over the two beams if we encounter a structure function that applies to the beam pair.

*(CCC Commands: procedures)+≡*

```

subroutine sf_list_register_user (sf_list, affects_beam, &
    model, flv, user_name, var_list)
type(sf_list_t), intent(inout) :: sf_list
logical, dimension(2), intent(in) :: affects_beam
type(model_t), intent(in), target :: model
type(flavor_t), dimension(2), intent(in) :: flv
type(string_t), intent(in) :: user_name
type(var_list_t), intent(in) :: var_list
type(sf_data_t), pointer :: sf_data
logical :: user_strfun_mapping
real(default) :: user_strfun_mapping_power
integer :: i
do i = 1, 2
    if (affects_beam(i)) then
        allocate (sf_data)
        call sf_data_init_user (sf_data, i, flv, user_name, model)
        call sf_list_append (sf_list, sf_data)
        if (all (sf_data_affects_beam (sf_data))) then
            if (.not. all (affects_beam)) call msg_fatal &
                ("User spectrum/structure function inconsistently applied")
            exit
        end if
    end if
end do
user_strfun_mapping = &
    var_list_get_lval (var_list, var_str ("?user_strfun_mapping"))
user_strfun_mapping_power = &

```



```

        var_list_get_rval (var_list, var_str ("user_strfun_mapping_power"))
if (all (affects_beam) .and. user_strfun_mapping) then
    if (all (sf_data_affects_beam (sf_data))) then
        call sf_data_setup_mapping &
            (sf_data, SFM_PAIR, &
             (/ sf_data_get_n_parameters (sf_data) - 1, &
              sf_data_get_n_parameters (sf_data) /), &
             user_strfun_mapping_power)
    else
        call sf_data_setup_mapping &
            (sf_data, SFM_PAIR, &
             (/ 0, sf_data_get_n_parameters (sf_data) /), &
             user_strfun_mapping_power)
    end if
end if
! No pair mapping
end subroutine sf_list_register_user

```

## Beam polarization

We define an assortment of containers for the options of the different beam polarization constructors.

```

<CCC Commands: types>+≡
type :: bp_circ_data_t
private
type(parse_node_t), pointer :: pn_fraction => null ()
real(default) :: fraction
end type bp_circ_data_t

type :: bp_trans_data_t
private
type(parse_node_t), pointer :: pn_fraction => null ()
type(parse_node_t), pointer :: pn_phi => null ()
real(default) :: fraction, phi
end type bp_trans_data_t

type :: bp_long_data_t
private
type(parse_node_t), pointer :: pn_fraction => null ()
real(default) :: fraction
end type bp_long_data_t

type :: bp_axis_data_t
private
type(parse_node_t), pointer :: pn_fraction => null ()
type(parse_node_t), pointer :: pn_theta => null ()
type(parse_node_t), pointer :: pn_phi => null ()
real(default) :: fraction, theta, phi
end type bp_axis_data_t

type :: bp_diag_data_t
private

```

```

        type(parse_node_p), dimension(:), allocatable :: pn_hel
        type(parse_node_p), dimension(:), allocatable :: pn_fraction
        integer, dimension(:), allocatable :: hel
        real(default), dimension(:), allocatable :: fraction
    end type bp_diag_data_t

    type :: bp_density_data_t
    private
        type(parse_node_t), pointer :: pn_d => null ()
        type(parse_node_t), pointer :: pn_nd => null ()
        real(default) :: d
        complex(default) :: nd
    end type bp_density_data_t

```

The actual scratch container for the command. A negative **n** means the structure is invalid, and a negative **type** tells the execution subprogram to disable beam polarization altogether.

```

<CCC Commands: types>+≡
    type :: cmd_beam_polarization_t
    private
        integer :: n = -1
        integer, dimension(2) :: type = -1
        type(bp_circ_data_t), dimension(:), pointer :: circ_data => null ()
        type(bp_trans_data_t), dimension(:), pointer :: trans_data => null ()
        type(bp_long_data_t), dimension(:), pointer :: long_data => null ()
        type(bp_axis_data_t), dimension(:), pointer :: axis_data => null ()
        type(bp_diag_data_t), dimension(:), pointer :: diag_data => null ()
        type(bp_density_data_t), dimension(:), pointer :: &
            density_data => null ()
        type(command_list_t), pointer :: options => null ()
        type(rt_data_t) :: local
        type(beam_polarization_t), dimension(:), pointer :: &
            beam_polarization => null ()
    end type cmd_beam_polarization_t

```

Finalize the container. We define a separate finalizer for each subcontainer.

```

<CCC Commands: procedures>+≡
    elemental subroutine bp_circ_data_final (d)
        type(bp_circ_data_t), intent(inout) :: d
    end subroutine bp_circ_data_final

    elemental subroutine bp_trans_data_final (d)
        type(bp_trans_data_t), intent(inout) :: d
    end subroutine bp_trans_data_final

    elemental subroutine bp_long_data_final (d)
        type(bp_long_data_t), intent(inout) :: d
    end subroutine bp_long_data_final

    elemental subroutine bp_axis_data_final (d)
        type(bp_axis_data_t), intent(inout) :: d
    end subroutine bp_axis_data_final

```

```

elemental subroutine bp_diag_data_final (d)
  type(bp_diag_data_t), intent(inout) :: d
  deallocate (d%pn_hel)
  deallocate (d%pn_fraction)
  if (allocated (d%hel)) deallocate (d%hel)
  if (allocated (d%fraction)) deallocate (d%fraction)
end subroutine bp_diag_data_final

elemental subroutine bp_density_data_final (d)
  type(bp_density_data_t), intent(inout) :: d
end subroutine bp_density_data_final

subroutine cmd_beam_polarization_final (bp)
  type(cmd_beam_polarization_t), intent(inout) :: bp
  if (associated (bp%circ_data)) then
    call bp_circ_data_final (bp%circ_data)
    deallocate (bp%circ_data)
  end if
  if (associated (bp%trans_data)) then
    call bp_trans_data_final (bp%trans_data)
    deallocate (bp%trans_data)
  end if
  if (associated (bp%long_data)) then
    call bp_long_data_final (bp%long_data)
    deallocate (bp%long_data)
  end if
  if (associated (bp%axis_data)) then
    call bp_axis_data_final (bp%axis_data)
    deallocate (bp%axis_data)
  end if
  if (associated (bp%diag_data)) then
    call bp_diag_data_final (bp%diag_data)
    deallocate (bp%diag_data)
  end if
  if (associated (bp%density_data)) then
    call bp_density_data_final (bp%density_data)
    deallocate (bp%density_data)
  end if
  if (associated (bp%options)) then
    call command_list_final (bp%options)
    deallocate (bp%options)
  end if
  if (associated (bp%beam_polarization)) deallocate (bp%beam_polarization)
  bp%type = -1
  bp%n = -1
end subroutine cmd_beam_polarization_final

```

Compile.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_beam_polarization_compile (bp, pn, global)
  type(cmd_beam_polarization_t), pointer, intent(inout) :: bp
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_list, pn_opts, pn_args, pn_entry

```

```

integer :: i, n
pn_list => parse_node_get_sub_ptr (pn, 3)
pn_opts => parse_node_get_sub_ptr (pn, 4)
allocate (bp)
call rt_data_local_init (bp%local, global)
if (associated (pn_opts)) then
    allocate (bp%options)
    call command_list_compile (bp%options, pn_opts, bp%local)
end if
if (parse_node_get_rule_key (pn_list) == "off") then
    bp%n = 0
    bp%type = -1
    return
end if
bp%n = parse_node_get_n_sub (pn_list)
pn_list => parse_node_get_sub_ptr (pn_list)
do i = 1, bp%n
    pn_args => parse_node_get_sub_ptr (pn_list, 2)
    select case (char (parse_node_get_rule_key (pn_list)))
        case ("none")
            bp%type(i) = BP_NONE
        case ("bp_circ")
            if (parse_node_get_n_sub (pn_args) /= 1) then
                call cmd_beam_polarization_final (bp)
                call msg_fatal &
                    ("syntax error: expecting 'circular (fraction)')")
                return
            end if
            bp%type(i) = BP_CIRC
            if (.not. associated (bp%circ_data)) &
                allocate (bp%circ_data(2))
            bp%circ_data(i)%pn_fraction => parse_node_get_sub_ptr (pn_args, 1)
        case ("bp_trans")
            if (parse_node_get_n_sub (pn_args) /= 2) then
                call cmd_beam_polarization_final (bp)
                call msg_fatal &
                    ("syntax error: expecting transverse (fraction, phi)')")
                return
            end if
            bp%type(i) = BP_TRANS
            if (.not. associated (bp%trans_data)) &
                allocate (bp%trans_data(2))
            bp%trans_data(i)%pn_fraction => parse_node_get_sub_ptr (pn_args, 1)
            bp%trans_data(i)%pn_phi => parse_node_get_sub_ptr (pn_args, 2)
        case ("bp_axis")
            if (parse_node_get_n_sub (pn_args) /= 3) then
                call cmd_beam_polarization_final (bp)
                call msg_fatal &
                    ("syntax error: expecting 'axis (fraction, theta, phi)')")
                return
            end if
            bp%type(i) = BP_AXIS
            if (.not. associated (bp%axis_data)) &
                allocate (bp%axis_data(2))
    end select
end do

```

```

        bp%axis_data(i)%pn_fraction => parse_node_get_sub_ptr (pn_args, 1)
        bp%axis_data(i)%pn_theta => parse_node_get_sub_ptr (pn_args, 2)
        bp%axis_data(i)%pn_phi => parse_node_get_sub_ptr (pn_args, 3)
    case ("bp_long")
        if (parse_node_get_n_sub (pn_args) /= 1) then
            call cmd_beam_polarization_final (bp)
            call msg_fatal &
                ("syntax error: expecting 'longitudinal (fraction)')")
            return
        end if
        bp%type(i) = BP_LONG
        if (.not. associated (bp%long_data)) &
            allocate (bp%long_data(2))
        bp%long_data(i)%pn_fraction => parse_node_get_sub_ptr (pn_args, 1)
    case ("bp_dens")
        if (parse_node_get_n_sub (pn_args) /= 2) then
            call cmd_beam_polarization_final (bp)
            call msg_fatal &
                ("syntax error: expecting 'density_matrix (a, b)')")
            return
        end if
        bp%type(i) = BP_DENSITY
        if (.not. associated (bp%density_data)) &
            allocate (bp%density_data (2))
        bp%density_data(i)%pn_d => parse_node_get_sub_ptr (pn_args, 1)
        bp%density_data(i)%pn_nd => parse_node_get_sub_ptr (pn_args, 2)
    case ("bp_diag")
        bp%type(i) = BP_DIAG
        n = parse_node_get_n_sub (pn_args)
        if (.not. associated (bp%diag_data)) &
            allocate (bp%diag_data(2))
        allocate (bp%diag_data(i)%pn_hel (n))
        allocate (bp%diag_data(i)%pn_fraction (n))
        allocate (bp%diag_data(i)%hel (n))
        allocate (bp%diag_data(i)%fraction (n))
        pn_entry => parse_node_get_sub_ptr (pn_args)
        n = 1
        do while (associated (pn_entry))
            bp%diag_data(i)%pn_hel(n)%ptr &
                => parse_node_get_sub_ptr (pn_entry, 1)
            bp%diag_data(i)%pn_fraction(n)%ptr &
                => parse_node_get_sub_ptr (pn_entry, 3)
            pn_entry => parse_node_get_next_ptr (pn_entry)
            n = n + 1
        end do
    case default
        call msg_bug ("cmd_beam_polarization_compile: invalid " &
            // "polarization type")
    end select
    pn_list => parse_node_get_next_ptr (pn_list)
end do

call rt_data_local_reset (bp%local)
end subroutine cmd_beam_polarization_compile

```

Execute.

*{CCC Commands: procedures}+≡*

```

subroutine cmd_beam_polarization_execute (bp, global)
  type(cmd_beam_polarization_t), pointer, intent(inout) :: bp
  type(rt_data_t), intent(inout), target :: global
  type(polarization_t), dimension(:), allocatable :: pol
  integer :: i, j, k, ulog
  ulog = logfile_unit ()
  call rt_data_link (bp%local, global)
  if (associated (bp%options)) &
    call command_list_execute (bp%options, bp%local)
  if (bp%n < 0) then
    call rt_data_restore (global, bp%local)
    return
  end if
  if (bp%type(1) < 0) then
    call rt_data_restore (global, bp%local)
    global%beam_polarization => null ()
    if (beam_data_are_valid (global%beam_data)) &
      call beam_data_kill_polarization (global%beam_data)
    if (global%environment /= CMD_BEAMS) call msg_message &
      ("beam polarization disabled")
    return
  end if
  if (.not. associated (bp%beam_polarization)) then
    allocate (bp%beam_polarization(bp%n))
  else
    do i = 1, bp%n
      call beam_polarization_final (bp%beam_polarization(i))
    end do
  end if
  do i = 1, bp%n
    select case (bp%type(i))
    case (BP_NONE, BP_TRIVIAL)
      if (bp%n == 2) then
        call beam_polarization_init_none (bp%beam_polarization(i))
      else
        call beam_polarization_init_trivial (bp%beam_polarization(i))
      end if
    case (BP_CIRC)
      bp%circ_data(i)%fraction = &
        eval_real (bp%circ_data(i)%pn_fraction, bp%local%var_list)
      call beam_polarization_init_circ (bp%beam_polarization(i), &
        bp%circ_data(i)%fraction)
    case (BP_TRANS)
      bp%trans_data(i)%fraction = &
        eval_real (bp%trans_data(i)%pn_fraction, bp%local%var_list)
      bp%trans_data(i)%phi = &
        eval_real (bp%trans_data(i)%pn_phi, bp%local%var_list)
      call beam_polarization_init_trans (bp%beam_polarization(i), &
        bp%trans_data(i)%fraction, bp%trans_data(i)%phi)
    case (BP_LONG)
      bp%long_data(i)%fraction = &

```

```

        eval_real (bp%long_data(i)%pn_fraction, bp%local%var_list)
    call beam_polarization_init_long (bp%beam_polarization(i), &
        bp%long_data(i)%fraction)
case (BP_AXIS)
    bp%axis_data(i)%fraction = &
        eval_real (bp%axis_data(i)%pn_fraction, bp%local%var_list)
    bp%axis_data(i)%theta = &
        eval_real (bp%axis_data(i)%pn_theta, bp%local%var_list)
    bp%axis_data(i)%phi = &
        eval_real (bp%axis_data(i)%pn_phi, bp%local%var_list)
    call beam_polarization_init_axis (bp%beam_polarization(i), &
        bp%axis_data(i)%fraction, bp%axis_data(i)%theta, &
        bp%axis_data(i)%phi)
case (BP_DENSITY)
    bp%density_data(i)%d = &
        eval_real (bp%density_data(i)%pn_d, bp%local%var_list)
    bp%density_data(i)%nd = &
        eval_cmlpx (bp%density_data(i)%pn_nd, bp%local%var_list)
    call beam_polarization_init_density (bp%beam_polarization(i), &
        bp%density_data(i)%d, bp%density_data(i)%nd)
case (BP_DIAG)
    do j = 1, size (bp%diag_data(i)%hel)
        bp%diag_data(i)%hel(j) = &
            eval_int (bp%diag_data(i)%pn_hel(j)%ptr, bp%local%var_list)
        bp%diag_data(i)%fraction(j) = &
            eval_real (bp%diag_data(i)%pn_fraction(j)%ptr, &
                bp%local%var_list)
        if (j > 1) then
            do k = 1, j - 1
                if (bp%diag_data(i)%hel(j) == bp%diag_data(i)%hel(k)) then
                    call msg_error ( &
                        "'diagonal_density (h1:f1 [, h2:f2, ...])': " &
                        // "h" // int2char(j) // " and h" // int2char(k) &
                        // " must not be equal")
                    call rt_data_restore (global, bp%local)
                    return
                end if
            end do
        end if
    end do
    call beam_polarization_init_diag (bp%beam_polarization(i), &
        bp%diag_data(i)%hel, bp%diag_data(i)%fraction)
case default
    call msg_bug ("cmd_beam_polarization_execute: " &
        // "unknown polarization type")
end select
if (global%environment /= CMD_BEAMS) then
    call msg_message &
        ("polarization of incoming particle " // int2char(i) // ":")
    call beam_polarization_write (bp%beam_polarization(i))
    call beam_polarization_write (bp%beam_polarization(i), ulog)
end if
end do

```

```

call rt_data_restore (global, bp%local)
global%beam_polarization => bp%beam_polarization
if (beam_data_are_valid (global%beam_data)) then
  if (beam_data_get_n_in (global%beam_data) /= bp%n) then
    call msg_error ("the number of incoming particles differs " &
      // "between beam and polarization setup - ignoring polarization")
  else
    allocate (pol (bp%n))
    do i = 1, bp%n
      pol(i) = beam_polarization2polarization &
        (bp%beam_polarization(i), global%beam_data%flv(i), &
          decay=(bp%n == 1))
    end do
    call beam_data_set_polarization (global%beam_data, pol)
  end if
else
  if (global%environment /= CMD_BEAMS) call msg_warning ( &
    "beam_polarization only works with a beam setup")
end if

end subroutine cmd_beam_polarization_execute

```

## Cuts

Define a cut expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the cut expression is used.

```

<Commands: types>+≡
  type, extends (command_t) :: cmd_cuts_t
  private
    type(parse_node_t), pointer :: pn_lexpr => null ()
  contains
    <Commands: cmd cuts: TBP>
  end type cmd_cuts_t

```

Output. Do not print the parse tree, since this may get cluttered. Just a message that cuts have been defined.

```

<Commands: cmd cuts: TBP>≡
  procedure :: write => cmd_cuts_write

<Commands: procedures>+≡
  subroutine cmd_cuts_write (cmd, unit, indent)
    class(cmd_cuts_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    write (u, "(1x,A)") "cuts: [defined]"
  end subroutine cmd_cuts_write

```



Compile. Simply store the parse (sub)tree.

```

<Commands: cmd cuts: TBP>+≡
  procedure :: compile => cmd_cuts_compile

<Commands: procedures>+≡
  subroutine cmd_cuts_compile (cmd, global)
    class(cmd_cuts_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    cmd%pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 3)
  end subroutine cmd_cuts_compile

```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

```

<Commands: cmd cuts: TBP>+≡
  procedure :: execute => cmd_cuts_execute

<Commands: procedures>+≡
  subroutine cmd_cuts_execute (cmd, global)
    class(cmd_cuts_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    global%pn%cuts_lexpr => cmd%pn_lexpr
  end subroutine cmd_cuts_execute

```

## General, Factorization and Renormalization Scales

Define a scale expression for either the renormalization or the factorization scale. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the expression is used.

```

<CCC Commands: types>+≡
  type :: cmd_scale_t
  private
    type(parse_node_t), pointer :: pn_expr => null ()
  end type cmd_scale_t

```

```

<CCC Commands: types>+≡
  type :: cmd_fac_scale_t
  private
    type(parse_node_t), pointer :: pn_expr => null ()
  end type cmd_fac_scale_t

```

```

<CCC Commands: types>+≡
  type :: cmd_ren_scale_t
  private
    type(parse_node_t), pointer :: pn_expr => null ()
  end type cmd_ren_scale_t

```

Compile. Simply store the parse (sub)tree.

```

<CCC Commands: procedures>+≡
  subroutine cmd_scale_compile (scale, pn)
    type(cmd_scale_t), pointer :: scale

```

```

    type(parse_node_t), intent(in), target :: pn
    allocate (scale)
    scale%pn_expr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_scale_compile

```

*<CCC Commands: procedures>+≡*

```

subroutine cmd_fac_scale_compile (scale, pn)
    type(cmd_fac_scale_t), pointer :: scale
    type(parse_node_t), intent(in), target :: pn
    allocate (scale)
    scale%pn_expr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_fac_scale_compile

```

*<CCC Commands: procedures>+≡*

```

subroutine cmd_ren_scale_compile (scale, pn)
    type(cmd_ren_scale_t), pointer :: scale
    type(parse_node_t), intent(in), target :: pn
    allocate (scale)
    scale%pn_expr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_ren_scale_compile

```

Instead of evaluating the scale expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_scale_execute (scale, global)
    type(cmd_scale_t), intent(inout), target :: scale
    type(rt_data_t), intent(inout), target :: global
    global%pn_scale_expr => scale%pn_expr
end subroutine cmd_scale_execute

```

*<CCC Commands: procedures>+≡*

```

subroutine cmd_fac_scale_execute (scale, global)
    type(cmd_fac_scale_t), intent(inout), target :: scale
    type(rt_data_t), intent(inout), target :: global
    global%pn_fac_scale_expr => scale%pn_expr
end subroutine cmd_fac_scale_execute

```

*<CCC Commands: procedures>+≡*

```

subroutine cmd_ren_scale_execute (scale, global)
    type(cmd_ren_scale_t), intent(inout), target :: scale
    type(rt_data_t), intent(inout), target :: global
    global%pn_ren_scale_expr => scale%pn_expr
end subroutine cmd_ren_scale_execute

```

## Weight

Define a weight expression. The weight is applied to a process to be integrated, event by event. We store the parse tree for the right-hand side instead of

compiling it. Compilation is deferred to the process environment where the expression is used.

```
<CCC Commands: types>+≡
  type :: cmd_weight_t
  private
    type(parse_node_t), pointer :: pn_expr => null ()
  end type cmd_weight_t
```

Compile. Simply store the parse (sub)tree.

```
<CCC Commands: procedures>+≡
  subroutine cmd_weight_compile (weight, pn)
    type(cmd_weight_t), pointer :: weight
    type(parse_node_t), intent(in), target :: pn
    allocate (weight)
    weight%pn_expr => parse_node_get_sub_ptr (pn, 3)
  end subroutine cmd_weight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

```
<CCC Commands: procedures>+≡
  subroutine cmd_weight_execute (weight, global)
    type(cmd_weight_t), intent(inout), target :: weight
    type(rt_data_t), intent(inout), target :: global
    global%pn_weight_expr => weight%pn_expr
  end subroutine cmd_weight_execute
```

## Selection

Define a selection expression. This is to applied upon simulation or event-file rescanning, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the environment where the expression is used.

```
<CCC Commands: types>+≡
  type :: cmd_selection_t
  private
    type(parse_node_t), pointer :: pn_expr => null ()
  end type cmd_selection_t
```

Compile. Simply store the parse (sub)tree.

```
<CCC Commands: procedures>+≡
  subroutine cmd_selection_compile (selection, pn)
    type(cmd_selection_t), pointer :: selection
    type(parse_node_t), intent(in), target :: pn
    allocate (selection)
    selection%pn_expr => parse_node_get_sub_ptr (pn, 3)
  end subroutine cmd_selection_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

```
<CCC Commands: procedures>+≡
subroutine cmd_selection_execute (selection, global)
  type(cmd_selection_t), intent(inout), target :: selection
  type(rt_data_t), intent(inout), target :: global
  global%pn_selection_lexpr => selection%pn_expr
end subroutine cmd_selection_execute
```

## Reweight

Define a reweight expression. This is to applied upon simulation or event-file rescanning, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the environment where the expression is used.

```
<CCC Commands: types>+≡
type :: cmd_reweight_t
  private
  type(parse_node_t), pointer :: pn_expr => null ()
end type cmd_reweight_t
```

Compile. Simply store the parse (sub)tree.

```
<CCC Commands: procedures>+≡
subroutine cmd_reweight_compile (reweight, pn)
  type(cmd_reweight_t), pointer :: reweight
  type(parse_node_t), intent(in), target :: pn
  allocate (reweight)
  reweight%pn_expr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_reweight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

```
<CCC Commands: procedures>+≡
subroutine cmd_reweight_execute (reweight, global)
  type(cmd_reweight_t), intent(inout), target :: reweight
  type(rt_data_t), intent(inout), target :: global
  global%pn_reweight_expr => reweight%pn_expr
end subroutine cmd_reweight_execute
```

## Integration

Integrate several processes, consecutively with identical parameters.

```
<Commands: types>+≡
type, extends (command_t) :: cmd_integrate_t
  private
  integer :: n_proc = 0
  type(string_t), dimension(:), allocatable :: process_id
  !   type(command_list_t), pointer :: options => null ()
  type(rt_data_t), pointer :: local
```

```

contains
  <Commands: cmd integrate: TBP>
end type cmd_integrate_t

```

Finalizer.

```

<CCC Commands: procedures>+≡
subroutine cmd_integrate_final (cmd)
  type(cmd_integrate_t), intent(inout) :: cmd
  if (associated (cmd%options)) then
    call command_list_final (cmd%options)
    deallocate (cmd%options)
  end if
end subroutine cmd_integrate_final

```

Output: we know the process IDs.

```

<Commands: cmd integrate: TBP>≡
  procedure :: write => cmd_integrate_write

<Commands: procedures>+≡
subroutine cmd_integrate_write (cmd, unit, indent)
  class(cmd_integrate_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "integrate ("
  do i = 1, cmd%n_proc
    if (i > 1) write (u, "(A,1x)", advance="no") ", "
    write (u, "(A)", advance="no") char (cmd%process_id(i))
  end do
  write (u, "(A)") ")"
end subroutine cmd_integrate_write

```

Compile.

```

<Commands: cmd integrate: TBP>+≡
  procedure :: compile => cmd_integrate_compile

<Commands: procedures>+≡
subroutine cmd_integrate_compile (cmd, global)
  class(cmd_integrate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc, pn_opt
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_opt => parse_node_get_next_ptr (pn_proclist)
!   call rt_data_local_init (cmd%local, global)
  cmd%local => global
!   if (associated (pn_opt)) then
!     allocate (cmd%options)
!     call command_list_compile (cmd%options, pn_opt, cmd%local)
!   end if
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (cmd%process_id (cmd%n_proc))

```

```

pn_proc => parse_node_get_sub_ptr (pn_proclist)
do i = 1, cmd%n_proc
    cmd%process_id(i) = parse_node_get_string (pn_proc)
    !      call var_list_init_process_results (global%var_list, &
    !      cmd%process_id (i))
    pn_proc => parse_node_get_next_ptr (pn_proc)
end do
!      call rt_data_local_reset (cmd%local)
end subroutine cmd_integrate_compile

```

Command execution. Integrate the process(es) with the predefined number of passes, iterations and calls. For structure functions, cuts, weight and scale, use local definitions if present; by default, the local definitions are initialized with the global ones.

(Re-)Integrating a decay process may invalidate a previous decay configuration, therefore we update the decay store.

Since the process acquires a snapshot of the variable list, so if the global list (or the local one) is deleted, this does no harm. This implies that later changes of the variable list do not affect the stored process.

*<Commands: cmd integrate: TBP>+≡*

```

procedure :: execute => cmd_integrate_execute

```

*<Commands: procedures>+≡*

```

subroutine cmd_integrate_execute (cmd, global)
    class(cmd_integrate_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    integer :: i
    !      call rt_data_link (cmd%local, global)
    !      if (associated (cmd%options)) then
    !          call command_list_execute (cmd%options, cmd%local)
    !      end if
    do i = 1, cmd%n_proc
        call integrate_process (cmd%process_id(i), cmd%local)
    end do
    !      call decay_store_update (cmd%process_id, verbose = .true.)
    !      call rt_data_restore (global, cmd%local)
end subroutine cmd_integrate_execute

```

## Testing the Matrix Element

This command takes a process and evaluates the matrix element a given number of times, randomly choosing an integration channel.

*<CCC Commands: types>+≡*

```

type :: cmd_me_test_t
    private
    type(string_t) :: process_id
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
end type cmd_me_test_t

```

Finalizer.

```
<CCC Commands: procedures>+≡
subroutine cmd_me_test_final (me_test)
  type(cmd_me_test_t), intent(inout) :: me_test
  if (associated (me_test%options)) then
    call command_list_final (me_test%options)
    deallocate (me_test%options)
  end if
end subroutine cmd_me_test_final
```

Compile.

```
<CCC Commands: procedures>+≡
subroutine cmd_me_test_compile (me_test, pn, global)
  type(cmd_me_test_t), pointer :: me_test
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc, pn_opt
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (pn, 2)
  pn_opt => parse_node_get_next_ptr (pn_proclist)
  allocate (me_test)
  call rt_data_local_init (me_test%local, global)
  if (associated (pn_opt)) then
    allocate (me_test%options)
    call command_list_compile (me_test%options, pn_opt, me_test%local)
  end if
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  me_test%process_id = parse_node_get_string (pn_proc)
  call rt_data_local_reset (me_test%local)
end subroutine cmd_me_test_compile
```

Command execution. Call the matrix element of a process a predefined number of times. This requires all the prerequisites of an integration, but it ignores all cuts, and it throws away the values. The chosen momenta are random.

```
<CCC Commands: procedures>+≡
subroutine cmd_me_test_execute (me_test, global)
  type(cmd_me_test_t), intent(inout), target :: me_test
  type(rt_data_t), intent(inout), target :: global
  call rt_data_link (me_test%local, global)
  if (associated (me_test%options)) then
    call command_list_execute (me_test%options, me_test%local)
  end if
  call me_test_process &
    (me_test%process_id, me_test%local, global%var_list)
  call rt_data_restore (global, me_test%local)
end subroutine cmd_me_test_execute
```

## Observables

Declare an observable. After the declaration, it can be used to record data, and at the end one can retrieve average and error.

```

<CCC Commands: types>+=
  type :: cmd_observable_t
    private
    logical :: use_id_expr = .false.
    type(string_t) :: id
    type(parse_node_t), pointer :: pn_id => null ()
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
  end type cmd_observable_t

```

Finalizer for the ID string expression

```

<CCC Commands: procedures>+=
  subroutine cmd_observable_final (observable)
    type(cmd_observable_t), intent(inout) :: observable
    if (associated (observable%options)) then
      call command_list_final (observable%options)
      deallocate (observable%options)
    end if
  end subroutine cmd_observable_final

```

Compile. Just record the observable ID.

```

<CCC Commands: procedures>+=
  subroutine cmd_observable_compile (observable, pn, global)
    type(cmd_observable_t), pointer :: observable
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_tag, pn_opt
    pn_tag => parse_node_get_sub_ptr (pn, 2)
    if (associated (pn_tag)) then
      pn_opt => parse_node_get_next_ptr (pn_tag)
    else
      pn_opt => null ()
    end if
    allocate (observable)
    call rt_data_local_init (observable%local, global)
    if (associated (pn_opt)) then
      allocate (observable%options)
      call command_list_compile (observable%options, pn_opt, observable%local)
    end if
    select case (char (parse_node_get_rule_key (pn_tag)))
    case ("analysis_id")
      observable%id = parse_node_get_string (pn_tag)
    case default
      observable%use_id_expr = .true.
      observable%pn_id => pn_tag
    end select
    call rt_data_local_reset (observable%local)
  end subroutine cmd_observable_compile

```

Command execution. This declares the observable and allocates it in the analysis store.

```

<CCC Commands: procedures>+=

```



```

subroutine cmd_observable_execute (observable, global)
  type(cmd_observable_t), intent(inout), target :: observable
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: label, obs_unit, title
  type(graph_options_t) :: graph_options
  call rt_data_link (observable%local, global)
  if (associated (observable%options)) then
    call command_list_execute (observable%options, observable%local)
  end if
  if (observable%use_id_expr) then
    observable%id = eval_string (observable%pn_id, observable%local%var_list)
  end if
  label = var_list_get_sval &
    (observable%local%var_list, var_str ("obs_label"))
  obs_unit = var_list_get_sval &
    (observable%local%var_list, var_str ("obs_unit"))
  title = var_list_get_sval (observable%local%var_list, var_str ("title"))
  call graph_options_init (graph_options)
  call analysis_init_observable &
    (observable%id, label, obs_unit, graph_options)
  call rt_data_restore (global, observable%local)
end subroutine cmd_observable_execute

```

## Histograms

Declare a histogram. At minimum, we have to set lower and upper bound and bin width.

```

<CCC Commands: types>+≡
  type :: cmd_histogram_t
  private
  type(string_t) :: id
  logical :: use_id_expr = .false.
  type(parse_node_t), pointer :: pn_id => null ()
  type(parse_node_t), pointer :: pn_lower_bound => null ()
  type(parse_node_t), pointer :: pn_upper_bound => null ()
  type(parse_node_t), pointer :: pn_bin_width => null ()
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_histogram_t

```

Finalizer for the eval trees.

```

<CCC Commands: procedures>+≡
  subroutine cmd_histogram_final (histogram)
    type(cmd_histogram_t), intent(inout) :: histogram
    if (associated (histogram%options)) then
      call command_list_final (histogram%options)
      deallocate (histogram%options)
    end if
  end subroutine cmd_histogram_final

```

Compile. Record the histogram ID and initialize lower, upper bound and bin width.

*<CCC Commands: procedures>+≡*

```
subroutine cmd_histogram_compile (histogram, pn, global)
  type(cmd_histogram_t), pointer :: histogram
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_tag, pn_args, pn_arg1, pn_arg2, pn_arg3
  type(parse_node_t), pointer :: pn_opt
  character(*), parameter :: e_illegal_use = &
    "illegal usage of 'histogram': insufficient number of arguments"
  pn_tag => parse_node_get_sub_ptr (pn, 2)
  pn_args => parse_node_get_next_ptr (pn_tag)
  if (associated (pn_args)) then
    pn_arg1 => parse_node_get_sub_ptr (pn_args)
    if (.not. associated (pn_arg1)) call msg_fatal (e_illegal_use)
    pn_arg2 => parse_node_get_next_ptr (pn_arg1)
    if (.not. associated (pn_arg2)) call msg_fatal (e_illegal_use)
    pn_arg3 => parse_node_get_next_ptr (pn_arg2)
    pn_opt => parse_node_get_next_ptr (pn_args)
  else
    pn_opt => null ()
  end if
  allocate (histogram)
  call rt_data_local_init (histogram%local, global)
  if (associated (pn_opt)) then
    allocate (histogram%options)
    call command_list_compile (histogram%options, pn_opt, histogram%local)
  end if
  select case (char (parse_node_get_rule_key (pn_tag)))
  case ("analysis_id")
    histogram%id = parse_node_get_string (pn_tag)
  case default
    histogram%use_id_expr = .true.
    histogram%pn_id => pn_tag
  end select
  histogram%pn_lower_bound => pn_arg1
  histogram%pn_upper_bound => pn_arg2
  histogram%pn_bin_width => pn_arg3
  call rt_data_local_reset (histogram%local)
end subroutine cmd_histogram_compile
```

Command execution. This declares the histogram and allocates it in the analysis store.

*<CCC Commands: procedures>+≡*

```
subroutine cmd_histogram_execute (histogram, global)
  type(cmd_histogram_t), intent(inout), target :: histogram
  type(rt_data_t), intent(inout), target :: global
  real(default) :: lower_bound, upper_bound, bin_width
  integer :: bin_number
  logical :: bin_width_is_used, normalize_bins
  type(string_t) :: obs_label, obs_unit
  type(graph_options_t) :: graph_options
```

```

type(drawing_options_t) :: drawing_options

call rt_data_link (histogram%local, global)
if (associated (histogram%options)) then
    call command_list_execute (histogram%options, histogram%local)
end if

if (histogram%use_id_expr) then
    histogram%id = eval_string (histogram%pn_id, histogram%local%var_list)
end if

lower_bound = eval_real (histogram%pn_lower_bound, histogram%local%var_list)
upper_bound = eval_real (histogram%pn_upper_bound, histogram%local%var_list)
if (associated (histogram%pn_bin_width)) then
    bin_width = eval_real (histogram%pn_bin_width, histogram%local%var_list)
    bin_width_is_used = .true.
else if (var_list_is_known &
    (histogram%local%var_list, var_str ("n_bins"))) then
    bin_number = var_list_get_ival &
    (histogram%local%var_list, var_str ("n_bins"))
    bin_width_is_used = .false.
else
    call msg_error ("Histogram '" // char (histogram%id) // &
    "' : neither bin width nor number is defined")
end if
normalize_bins = var_list_get_lval &
    (histogram%local%var_list, var_str ("?normalize_bins"))
obs_label = var_list_get_sval &
    (histogram%local%var_list, var_str ("obs_label"))
obs_unit = var_list_get_sval &
    (histogram%local%var_list, var_str ("obs_unit"))

call graph_options_init (graph_options)
call set_graph_options (graph_options, histogram%local%var_list)
call drawing_options_init_histogram (drawing_options)
call set_drawing_options (drawing_options, histogram%local%var_list)

if (bin_width_is_used) then
    call analysis_init_histogram &
    (histogram%id, lower_bound, upper_bound, bin_width, &
    normalize_bins, &
    obs_label, obs_unit, &
    graph_options, drawing_options)
else
    call analysis_init_histogram &
    (histogram%id, lower_bound, upper_bound, bin_number, &
    normalize_bins, &
    obs_label, obs_unit, &
    graph_options, drawing_options)
end if
call rt_data_restore (global, histogram%local)

end subroutine cmd_histogram_execute

```

Set the graph options from a variable list.

*(CCC Commands: procedures)+≡*

```

subroutine set_graph_options (gro, var_list)
  type(graph_options_t), intent(inout) :: gro
  type(var_list_t), intent(in) :: var_list
  call graph_options_set (gro, title = &
    var_list_get_sval (var_list, var_str ("title")))
  call graph_options_set (gro, description = &
    var_list_get_sval (var_list, var_str ("description")))
  call graph_options_set (gro, x_label = &
    var_list_get_sval (var_list, var_str ("x_label")))
  call graph_options_set (gro, y_label = &
    var_list_get_sval (var_list, var_str ("y_label")))
  call graph_options_set (gro, width_mm = &
    var_list_get_ival (var_list, var_str ("graph_width_mm")))
  call graph_options_set (gro, height_mm = &
    var_list_get_ival (var_list, var_str ("graph_height_mm")))
  call graph_options_set (gro, x_log = &
    var_list_get_lval (var_list, var_str ("x_log")))
  call graph_options_set (gro, y_log = &
    var_list_get_lval (var_list, var_str ("y_log")))
  if (var_list_is_known (var_list, var_str ("x_min"))) &
    call graph_options_set (gro, x_min = &
      var_list_get_rval (var_list, var_str ("x_min")))
  if (var_list_is_known (var_list, var_str ("x_max"))) &
    call graph_options_set (gro, x_max = &
      var_list_get_rval (var_list, var_str ("x_max")))
  if (var_list_is_known (var_list, var_str ("y_min"))) &
    call graph_options_set (gro, y_min = &
      var_list_get_rval (var_list, var_str ("y_min")))
  if (var_list_is_known (var_list, var_str ("y_max"))) &
    call graph_options_set (gro, y_max = &
      var_list_get_rval (var_list, var_str ("y_max")))
  call graph_options_set (gro, gmlcode_bg = &
    var_list_get_sval (var_list, var_str ("gmlcode_bg")))
  call graph_options_set (gro, gmlcode_fg = &
    var_list_get_sval (var_list, var_str ("gmlcode_fg")))
end subroutine set_graph_options

```

Set the drawing options from a variable list.

*(CCC Commands: procedures)+≡*

```

subroutine set_drawing_options (dro, var_list)
  type(drawing_options_t), intent(inout) :: dro
  type(var_list_t), intent(in) :: var_list
  if (var_list_is_known (var_list, var_str ("draw_histogram"))) then
    if (var_list_get_lval (var_list, var_str ("draw_histogram"))) then
      call drawing_options_set (dro, with_hbars = .true.)
    else
      call drawing_options_set (dro, with_hbars = .false., &
        with_base = .false., fill = .false., piecewise = .false.)
    end if
  end if
  if (var_list_is_known (var_list, var_str ("draw_base"))) then

```

```

        if (var_list_get_lval (var_list, var_str ("?draw_base"))) then
            call drawing_options_set (dro, with_base = .true.)
        else
            call drawing_options_set (dro, with_base = .false., fill = .false.)
        end if
    end if
end if
if (var_list_is_known (var_list, var_str ("?draw_pieewise"))) then
    if (var_list_get_lval (var_list, var_str ("?draw_pieewise"))) then
        call drawing_options_set (dro, pieewise = .true.)
    else
        call drawing_options_set (dro, pieewise = .false.)
    end if
end if
if (var_list_is_known (var_list, var_str ("?fill_curve"))) then
    if (var_list_get_lval (var_list, var_str ("?fill_curve"))) then
        call drawing_options_set (dro, fill = .true., with_base = .true.)
    else
        call drawing_options_set (dro, fill = .false.)
    end if
end if
if (var_list_is_known (var_list, var_str ("?draw_curve"))) then
    if (var_list_get_lval (var_list, var_str ("?draw_curve"))) then
        call drawing_options_set (dro, draw = .true.)
    else
        call drawing_options_set (dro, draw = .false.)
    end if
end if
if (var_list_is_known (var_list, var_str ("?draw_errors"))) then
    if (var_list_get_lval (var_list, var_str ("?draw_errors"))) then
        call drawing_options_set (dro, err = .true.)
    else
        call drawing_options_set (dro, err = .false.)
    end if
end if
if (var_list_is_known (var_list, var_str ("?draw_symbols"))) then
    if (var_list_get_lval (var_list, var_str ("?draw_symbols"))) then
        call drawing_options_set (dro, symbols = .true.)
    else
        call drawing_options_set (dro, symbols = .false.)
    end if
end if
if (var_list_is_known (var_list, var_str ("$fill_options"))) then
    call drawing_options_set (dro, fill_options = &
        var_list_get_sval (var_list, var_str ("$fill_options")))
end if
if (var_list_is_known (var_list, var_str ("$draw_options"))) then
    call drawing_options_set (dro, draw_options = &
        var_list_get_sval (var_list, var_str ("$draw_options")))
end if
if (var_list_is_known (var_list, var_str ("$err_options"))) then
    call drawing_options_set (dro, err_options = &
        var_list_get_sval (var_list, var_str ("$err_options")))
end if
if (var_list_is_known (var_list, var_str ("$symbol"))) then

```

```

        call drawing_options_set (dro, symbol = &
            var_list_get_sval (var_list, var_str ("symbol")))
    end if
    if (var_list_is_known (var_list, var_str ("gmlcode_bg"))) then
        call drawing_options_set (dro, gmlcode_bg = &
            var_list_get_sval (var_list, var_str ("gmlcode_bg")))
    end if
    if (var_list_is_known (var_list, var_str ("gmlcode_fg"))) then
        call drawing_options_set (dro, gmlcode_fg = &
            var_list_get_sval (var_list, var_str ("gmlcode_fg")))
    end if
end subroutine set_drawing_options

```

## Plots

Declare a plot. No mandatory arguments, just options.

```

<CCC Commands: types>+≡
type :: cmd_plot_t
private
type(string_t) :: id
logical :: use_id_expr = .false.
type(parse_node_t), pointer :: pn_id => null ()
type(command_list_t), pointer :: options => null ()
type(rt_data_t) :: local
end type cmd_plot_t

```

Finalizer for the eval trees.

```

<CCC Commands: procedures>+≡
subroutine cmd_plot_final (plot)
type(cmd_plot_t), intent(inout) :: plot
if (associated (plot%options)) then
    call command_list_final (plot%options)
    deallocate (plot%options)
end if
end subroutine cmd_plot_final

```

Compile. Record the plot ID and initialize lower, upper bound and bin width.

```

<CCC Commands: procedures>+≡
subroutine cmd_plot_compile (plot, pn, global)
type(cmd_plot_t), pointer :: plot
type(parse_node_t), intent(in), target :: pn
type(rt_data_t), intent(in), target :: global
type(parse_node_t), pointer :: pn_tag
type(parse_node_t), pointer :: pn_opt
pn_tag => parse_node_get_sub_ptr (pn, 2)
pn_opt => parse_node_get_next_ptr (pn_tag)
allocate (plot)
call cmd_plot_init (plot, pn_tag, pn_opt, global)
end subroutine cmd_plot_compile

```

This init routine is separated because it is reused below for graph initialization.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_plot_init (plot, pn_tag, pn_opt, global)
  type(cmd_plot_t), intent(out) :: plot
  type(parse_node_t), intent(in), pointer :: pn_tag, pn_opt
  type(rt_data_t), intent(in), target :: global
  call rt_data_local_init (plot%local, global)
  if (associated (pn_opt)) then
    allocate (plot%options)
    call command_list_compile (plot%options, pn_opt, plot%local)
  end if
  select case (char (parse_node_get_rule_key (pn_tag)))
  case ("analysis_id")
    plot%id = parse_node_get_string (pn_tag)
  case default
    plot%use_id_expr = .true.
    plot%pn_id => pn_tag
  end select
  call rt_data_local_reset (plot%local)
end subroutine cmd_plot_init

```

Command execution. This declares the plot and allocates it in the analysis store.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_plot_execute (plot, global)
  type(cmd_plot_t), intent(inout), target :: plot
  type(rt_data_t), intent(inout), target :: global
  type(graph_options_t) :: graph_options
  type(drawing_options_t) :: drawing_options

  call rt_data_link (plot%local, global)
  if (associated (plot%options)) then
    call command_list_execute (plot%options, plot%local)
  end if

  if (plot%use_id_expr) then
    plot%id = eval_string (plot%pn_id, plot%local%var_list)
  end if

  call graph_options_init (graph_options)
  call set_graph_options (graph_options, plot%local%var_list)
  call drawing_options_init_plot (drawing_options)
  call set_drawing_options (drawing_options, plot%local%var_list)

  call analysis_init_plot (plot%id, graph_options, drawing_options)

  call rt_data_restore (global, plot%local)

end subroutine cmd_plot_execute

```

## Graphs

Declare a graph. The graph is defined in terms of its contents. Both the graph and its contents may carry options.

The graph object contains its own ID as well as the IDs of its elements. For the elements, we reuse the `cmd_plot_t` defined above.

```
<CCC Commands: types>+≡
  type :: cmd_graph_t
    private
    type(string_t) :: id
    type(parse_node_t), pointer :: pn_id
    logical :: use_id_expr = .false.
    integer :: n_elements = 0
    type(cmd_plot_t), dimension(:), allocatable :: el
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
  end type cmd_graph_t
```

Finalizer for the eval trees.

```
<CCC Commands: procedures>+≡
  subroutine cmd_graph_final (graph)
    type(cmd_graph_t), intent(inout) :: graph
    integer :: i
    if (allocated (graph%el)) then
      do i = 1, size (graph%el)
        call cmd_plot_final (graph%el(i))
      end do
      deallocate (graph%el)
    end if
    if (associated (graph%options)) then
      call command_list_final (graph%options)
      deallocate (graph%options)
    end if
  end subroutine cmd_graph_final
```

Compile. Record the graph ID and initialize lower, upper bound and bin width. For compiling the graph element syntax, we use part of the `cmd_plot_t` compiler.

```
<CCC Commands: procedures>+≡
  subroutine cmd_graph_compile (graph, pn, global)
    type(cmd_graph_t), pointer :: graph
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_term, pn_tag, pn_opt, pn_def, pn_app
    integer :: i

    pn_term => parse_node_get_sub_ptr (pn, 2)
    pn_tag => parse_node_get_sub_ptr (pn_term)
    pn_opt => parse_node_get_next_ptr (pn_tag)
    allocate (graph)
    call rt_data_local_init (graph%local, global)
    if (associated (pn_opt)) then
      allocate (graph%options)
```



```

        call command_list_compile (graph%options, pn_opt, graph%local)
    end if
    select case (char (parse_node_get_rule_key (pn_tag)))
    case ("analysis_id")
        graph%id = parse_node_get_string (pn_tag)
    case default
        graph%use_id_expr = .true.
        graph%pn_id => pn_tag
    end select
    pn_def => parse_node_get_next_ptr (pn_term, 2)
    graph%n_elements = parse_node_get_n_sub (pn_def)
    call rt_data_local_reset (graph%local)

    allocate (graph%el (graph%n_elements))
    pn_term => parse_node_get_sub_ptr (pn_def)
    pn_tag => parse_node_get_sub_ptr (pn_term)
    pn_opt => parse_node_get_next_ptr (pn_tag)
    call cmd_plot_init (graph%el(1), pn_tag, pn_opt, global)
    pn_app => parse_node_get_next_ptr (pn_term)
    do i = 2, graph%n_elements
        pn_term => parse_node_get_sub_ptr (pn_app, 2)
        pn_tag => parse_node_get_sub_ptr (pn_term)
        pn_opt => parse_node_get_next_ptr (pn_tag)
        call cmd_plot_init (graph%el(i), pn_tag, pn_opt, global)
        pn_app => parse_node_get_next_ptr (pn_app)
    end do

end subroutine cmd_graph_compile

```

Command execution. This declares the graph, allocates it in the analysis store, and copies the graph elements.

For the graph, we set graph and default drawing options. For the elements, we reset individual drawing options.

This accesses internals of the contained elements of type `cmd_plot_t`, see above. We might disentangle such an interdependency when this code is rewritten using proper type extension.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_graph_execute (graph, global)
    type(cmd_graph_t), intent(inout), target :: graph
    type(rt_data_t), intent(inout), target :: global
    type(graph_options_t) :: graph_options
    type(drawing_options_t) :: drawing_options
    integer :: i, type

    call rt_data_link (graph%local, global)
    if (associated (graph%options)) then
        call command_list_execute (graph%options, graph%local)
    end if

    if (graph%use_id_expr) then
        graph%id = eval_string (graph%pn_id, graph%local%var_list)
    end if

```

```

call graph_options_init (graph_options)
call set_graph_options (graph_options, graph%local%var_list)
call analysis_init_graph (graph%id, graph%n_elements, graph_options)

do i = 1, graph%n_elements
  call rt_data_link (graph%el(i)%local, global)
  if (associated (graph%el(i)%options)) then
    call command_list_execute (graph%el(i)%options, graph%el(i)%local)
  end if
  if (graph%el(i)%use_id_expr) then
    graph%el(i)%id = &
      eval_string (graph%el(i)%pn_id, graph%el(i)%local%var_list)
  end if
  type = analysis_store_get_object_type (graph%el(i)%id)
  select case (type)
  case (AN_HISTOGRAM)
    call drawing_options_init_histogram (drawing_options)
  case (AN_PLOT)
    call drawing_options_init_plot (drawing_options)
  end select
  call set_drawing_options (drawing_options, graph%local%var_list)
  if (associated (graph%el(i)%options)) then
    call set_drawing_options (drawing_options, graph%el(i)%local%var_list)
  end if
  call analysis_fill_graph (graph%id, i, graph%el(i)%id, drawing_options)
  call rt_data_restore (global, graph%el(i)%local)
end do

call rt_data_restore (global, graph%local)

end subroutine cmd_graph_execute

```

## Analysis

Hold the analysis ID either as a string or as an expression:

```

<CCC Commands: types>+≡
  type :: analysis_id_t
  type(string_t) :: tag
  type(parse_node_t), pointer :: pn_sexpr => null ()
end type analysis_id_t

```

Define the analysis expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the analysis expression is used.

```

<CCC Commands: types>+≡
  type :: cmd_analysis_t
  private
  type(parse_node_t), pointer :: pn_lexpr => null ()
end type cmd_analysis_t

```

Compile. Simply store the parse (sub)tree.

```
<CCC Commands: procedures>+≡
subroutine cmd_analysis_compile (analysis, pn)
  type(cmd_analysis_t), pointer :: analysis
  type(parse_node_t), intent(in), target :: pn
  allocate (analysis)
  analysis%pn_lexpr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_analysis_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

```
<CCC Commands: procedures>+≡
subroutine cmd_analysis_execute (analysis, global)
  type(cmd_analysis_t), intent(inout), target :: analysis
  type(rt_data_t), intent(inout), target :: global
  global%pn_analysis_lexpr => analysis%pn_lexpr
end subroutine cmd_analysis_execute
```

## Clear analysis objects

The syntax is analogous to `compile_analysis`. Each argument clears a particular analysis object. No argument clears the whole analysis store. The objects are cleared, but not deleted.

```
<CCC Commands: types>+≡
type :: cmd_clear_t
  private
  integer :: n_args = 0
  type(string_t), dimension(:), allocatable :: id
  logical, dimension(:), allocatable :: use_id_expr
  type(parse_node_p), dimension(:), allocatable :: pn_id
end type cmd_clear_t
```

Finalizer for the eval trees.

```
<CCC Commands: procedures>+≡
subroutine cmd_clear_final (clear)
  type(cmd_clear_t), intent(inout) :: clear
  if (allocated (clear%pn_id)) deallocate (clear%pn_id)
end subroutine cmd_clear_final
```

Compile. Record the clear ID and initialize lower, upper bound and bin width.

```
<CCC Commands: procedures>+≡
subroutine cmd_clear_compile (clear, pn, global)
  type(cmd_clear_t), pointer :: clear
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_args, pn_tag
  type(string_t) :: key
  integer :: i
  pn_args => parse_node_get_sub_ptr (pn, 2)
  allocate (clear)
```

```

if (associated (pn_args)) then
  clear%n_args = parse_node_get_n_sub (pn_args)
  allocate (clear%id (clear%n_args), clear%use_id_expr (clear%n_args))
  clear%use_id_expr = .false.
  allocate (clear%pn_id (clear%n_args))
  pn_tag => parse_node_get_sub_ptr (pn_args)
  i = 1
  do while (associated (pn_tag))
    key = parse_node_get_rule_key (pn_tag)
    select case (char (key))
      case ("iterations", "cuts", "weight", "scale", "factorization_scale", &
            "renormalization_scale", "analysis", "expect")
        clear%id(i) = key
      case ("analysis_id")
        clear%id(i) = parse_node_get_string (pn_tag)
      case default
        clear%pn_id(i)%ptr => pn_tag
    end select
    i = i + 1
    pn_tag => parse_node_get_next_ptr (pn_tag)
  end do
end if
end subroutine cmd_clear_compile

```

Command execution.

*{CCC Commands: procedures}+=*

```

subroutine cmd_clear_execute (clear, global)
  type(cmd_clear_t), intent(inout), target :: clear
  type(rt_data_t), intent(inout), target :: global
  integer :: i
  if (clear%n_args == 0) then
    call analysis_clear ()
    call msg_message ("Cleared all analysis objects")
  else
    do i = 1, clear%n_args
      select case (char (clear%id(i)))
        case ("iterations")
          call iterations_list_clear (global%it_list)
          call msg_message ("Cleared iteration list setup")
        case ("cuts")
          global%pn_cuts_lexpr => null ()
          call msg_message ("Cleared cut setup")
        case ("weight")
          global%pn_weight_expr => null ()
          call msg_message ("Cleared integration weight setup")
        case ("scale")
          global%pn_scale_expr => null ()
          call msg_message ("Cleared general scale setup")
        case ("factorization_scale")
          global%pn_fac_scale_expr => null ()
          call msg_message ("Cleared factorization scale setup")
        case ("renormalization_scale")
          global%pn_ren_scale_expr => null ()
          call msg_message ("Cleared renormalization scale setup")
      end select
    end do
  end if
end subroutine cmd_clear_execute

```

```

        case ("analysis")
            global%pn_analysis_lexpr => null ()
            call msg_message ("Cleared analysis setup")
        case ("expect")
            call expect_clear ()
            call msg_message ("Cleared counters of value checks")
        case default
            if (clear%use_id_expr(i)) then
                clear%id(i) = eval_string (clear%pn_id(i)%ptr, global%var_list)
            end if
            call analysis_clear (clear%id(i))
            call msg_message ("Cleared analysis object '" &
                // char (clear%id(i)) // "'")
        end select
    end do
end if
end subroutine cmd_clear_execute

```

## Write histograms and plots

The data type encapsulating the command:

```

<CCC Commands: types>+≡
type :: cmd_write_analysis_t
    type(analysis_id_t), dimension(:), allocatable :: id
    type(string_t), dimension(:), allocatable :: tag
    type(rt_data_t) :: local
    type(command_list_t), pointer :: options => null ()
end type cmd_write_analysis_t

<CCC Commands: procedures>+≡
subroutine cmd_write_analysis_final (write_analysis)
    type(cmd_write_analysis_t), intent(inout) :: write_analysis
    if (allocated (write_analysis%id)) deallocate (write_analysis%id)
    if (allocated (write_analysis%tag)) deallocate (write_analysis%tag)
    if (associated (write_analysis%options)) then
        call command_list_final (write_analysis%options)
        deallocate (write_analysis%options)
    end if
end subroutine cmd_write_analysis_final

```

Compile.

```

<CCC Commands: procedures>+≡
subroutine cmd_write_analysis_compile (write_analysis, pn, global)
    type(cmd_write_analysis_t), intent(inout), pointer :: write_analysis
    type(parse_node_t), intent(in) :: pn
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_clause, pn_opts, pn_args, pn_id
    integer :: n, i
    pn_clause => parse_node_get_sub_ptr (pn)
    pn_args => parse_node_get_sub_ptr (pn_clause, 2)
    pn_opts => parse_node_get_next_ptr (pn_clause)

```

```

allocate (write_analysis)
call rt_data_local_init (write_analysis%local, global)
if (associated (pn_opts)) then
    allocate (write_analysis%options)
    call command_list_compile &
        (write_analysis%options, pn_opts, write_analysis%local)
end if
if (associated (pn_args)) then
    n = parse_node_get_n_sub (pn_args)
    allocate (write_analysis%id (n))
    do i = 1, n
        pn_id => parse_node_get_sub_ptr (pn_args, i)
        if (char (parse_node_get_rule_key (pn_id)) == "analysis_id") then
            write_analysis%id(i)%tag = parse_node_get_string (pn_id)
        else
            write_analysis%id(i)%pn_sexpr => pn_id
        end if
    end do
else
    allocate (write_analysis%id (0))
end if
call rt_data_local_reset (write_analysis%local)
end subroutine cmd_write_analysis_compile

```

The output format for real data values:

*(Limits: public parameters)+≡*

```

character(*), parameter, public :: &
    DEFAULT_ANALYSIS_FILENAME = "whizard_analysis.dat"
character(len=1), dimension(2), parameter, public :: &
    FORBIDDEN_ENDINGS1 = (/ "o", "a" /)
character(len=2), dimension(5), parameter, public :: &
    FORBIDDEN_ENDINGS2 = (/ "mp", "ps", "vg", "lo", "la" /)
character(len=3), dimension(13), parameter, public :: &
    FORBIDDEN_ENDINGS3 = (/ "aux", "dvi", "evx", "f03", "f90", "log", &
        "ltp", "mpx", "pdf", "phs", "sin", "tex", "vbg" /)

```

Execute. If the `data_filename` optional argument is present, this is called from `cmd_compile_analysis_execute`, which needs the file name for further processing, and requires the default format. Otherwise, we use the parameters and macros for custom data processing.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_write_analysis_execute &
    (write_analysis, global, data_file, write_yerr, write_xerr)
type(cmd_write_analysis_t), intent(inout), target :: write_analysis
type(rt_data_t), intent(inout), target :: global
type(string_t), intent(out), optional :: data_file
logical, intent(out), optional :: write_yerr, write_xerr
type(string_t) :: defaultfile, file
logical :: keep_open, custom, header, columns
type(string_t) :: comment_prefix, separator, extension
integer :: i, type
type(ifile_t) :: ifile
logical :: one_file, has_writer

```

```

type(analysis_iterator_t) :: iterator
type(rt_data_t), target :: sandbox
type(command_list_t) :: writer

call rt_data_link (write_analysis%local, global)
if (associated (write_analysis%options)) then
    call command_list_execute (write_analysis%options, write_analysis%local)
end if

defaultfile = var_list_get_sval (write_analysis%local%var_list, &
    var_str ("out_file"))
if (present (data_file)) then
    if (defaultfile == "" .or. defaultfile == ".") then
        defaultfile = DEFAULT_ANALYSIS_FILENAME
    else
        if (scan(".", defaultfile) > 0) then
            call split (defaultfile, extension, ".", back=.true.)
            if (any (lower_case (char(extension)) == FORBIDDEN_ENDINGS1) .or. &
                any (lower_case (char(extension)) == FORBIDDEN_ENDINGS2) .or. &
                any (lower_case (char(extension)) == FORBIDDEN_ENDINGS3)) &
                call msg_fatal ("The ending " // char(extension) // &
                    " is internal and not allowed as data file.")
            if (extension /= "") then
                if (defaultfile /= "") then
                    defaultfile = defaultfile // "." // extension
                else
                    defaultfile = "whizard_analysis." // extension
                end if
            else
                defaultfile = defaultfile // ".dat"
            endif
        else
            defaultfile = defaultfile // ".dat"
        end if
    end if
    data_file = defaultfile
end if
one_file = defaultfile /= ""
if (one_file) then
    file = defaultfile
    keep_open = file_list_is_open (global%out_files, file, &
        action = "write")
    if (keep_open) then
        if (present (data_file)) then
            call msg_fatal ("Compiling analysis: File '" &
                // char (data_file) &
                // "' can't be used, it is already open.")
        else
            call msg_message ("Appending analysis data to file '" &
                // char (file) // "'")
        end if
    else
        !!! JRR: WK plase check
        !!! Left-over debug output from release 2.1.1
    end if
end if

```

```

        !!! print *, "calling file_list_open from cmd_write_analysis_execute"
        call file_list_open (global%out_files, file, &
            action = "write", status = "replace", position = "asis")
        call msg_message ("Writing analysis data to file '" &
            // char (file) // "'")
    end if
end if

if (present (data_file)) then
    custom = .false.
else
    custom = var_list_get_lval (write_analysis%local%var_list, &
        var_str ("?out_custom"))
end if
comment_prefix = var_list_get_sval (write_analysis%local%var_list, &
    var_str ("?out_comment"))
separator = var_list_get_sval (write_analysis%local%var_list, &
    var_str ("?out_separator"))
columns = var_list_get_lval (write_analysis%local%var_list, &
    var_str ("?out_columns"))
header = var_list_get_lval (write_analysis%local%var_list, &
    var_str ("?out_header"))
call get_analysis_tags (write_analysis%tag, write_analysis%id, &
    write_analysis%local%var_list)
if (present (write_yerr)) then
    write_yerr = var_list_get_lval (write_analysis%local%var_list, &
        var_str ("?out_yerr"))
end if
if (present (write_xerr)) then
    write_xerr = var_list_get_lval (write_analysis%local%var_list, &
        var_str ("?out_xerr"))
end if

do i = 1, size (write_analysis%tag)
    if (custom) then
        type = analysis_store_get_object_type (write_analysis%tag(i))
        call compile_writer (writer, type, has_writer)
        if (has_writer) then
            call rt_data_link (sandbox, write_analysis%local)
        end if
        if (.not. one_file) then
            file = write_analysis%tag(i) // ".dat"
            call file_list_open (global%out_files, file, &
                action = "write", status = "replace", position = "asis")
            call msg_message ("Writing analysis data to file '" &
                // char (file) // "'")
        end if
        if (header) then
            call analysis_get_header &
                (write_analysis%tag(i), ifile, comment_prefix)
            call file_list_write (global%out_files, file, ifile)
            call ifile_final (ifile)
        end if
        call analysis_init_iterator (write_analysis%tag(i), iterator)
    end if
end do

```



```

do while (analysis_iterator_is_valid (iterator))
  call write_data (iterator, &
    writer, sandbox, file, columns, separator)
  call analysis_iterator_advance (iterator)
end do
if (.not. one_file) then
  call file_list_close (global%out_files, file)
end if
if (has_writer) then
  call rt_data_restore (write_analysis%local, sandbox)
  call var_list_final (sandbox%var_list)
  call command_list_final (writer)
end if
else
  call file_list_write_analysis &
    (global%out_files, file, write_analysis%tag(i))
end if
end do

if (one_file .and. .not. keep_open) then
  call file_list_close (global%out_files, file)
end if
call rt_data_restore (global, write_analysis%local)

contains

subroutine get_analysis_tags (analysis_tag, id, var_list)
  type(string_t), dimension(:), intent(out), allocatable :: analysis_tag
  type(analysis_id_t), dimension(:), intent(in) :: id
  type(var_list_t), intent(in), target :: var_list
  if (size (id) /= 0) then
    allocate (analysis_tag (size (id)))
    do i = 1, size (id)
      if (associated (id(i)%pn_sexpr)) then
        analysis_tag(i) = eval_string (id(i)%pn_sexpr, var_list)
      else
        analysis_tag(i) = id(i)%tag
      end if
    end do
  else
    call analysis_store_get_ids (analysis_tag)
  end if
end subroutine get_analysis_tags

subroutine compile_writer (writer, type, has_writer)
  type(command_list_t), intent(out) :: writer
  integer, intent(in) :: type
  logical, intent(out) :: has_writer
  select case (type)
  case (AN_HISTOGRAM)
    has_writer = associated (write_analysis%local%pn_histogram_writer)
    if (has_writer) then
      call histogram_writer_compile (writer, &
        write_analysis%local%pn_histogram_writer, sandbox, &

```

```

        write_analysis%local)
    end if
case (AN_PLOT)
    has_writer = associated (write_analysis%local%pn_plot_writer)
    if (has_writer) then
        call plot_writer_compile (writer, &
            write_analysis%local%pn_plot_writer, sandbox, &
            write_analysis%local)
    end if
case default
    has_writer = .false.
    call msg_error ("Custom output format " &
        // "is applicable only to elementary histograms and plots")
end select
end subroutine compile_writer

subroutine write_data (iterator, writer, sandbox, file, columns, separator)
    type(analysis_iterator_t), intent(inout) :: iterator
    type(command_list_t), intent(in) :: writer
    type(rt_data_t), intent(inout), target :: sandbox
    type(string_t), intent(in) :: file
    logical, intent(in) :: columns
    type(string_t), intent(in) :: separator
    real(default) :: x, y, yerr, xerr, excess, width
    integer :: bin_index, n_bins, point_index, n_points
    character(*), parameter :: &
        OUT_REAL_FMT = '(' // HISTOGRAM_DATA_FORMAT // ')'
    select case (analysis_iterator_get_type (iterator))
case (AN_HISTOGRAM)
    call analysis_iterator_get_data (iterator, &
        x = x, y = y, yerr = yerr, width = width, excess = excess, &
        index = bin_index, n_total = n_bins)
    if (has_writer) then
        call histogram_writer_execute (writer, sandbox, file, &
            x, width, y, yerr, excess, bin_index, n_bins)
    else if (columns) then
        call file_list_write (global%out_files, file, &
            real2string (x, OUT_REAL_FMT) // separator // &
            real2string (y, OUT_REAL_FMT) // separator // &
            real2string (yerr, OUT_REAL_FMT))
    else
        call file_list_write (global%out_files, file, &
            real2string (x) // separator // &
            real2string (y) // separator // &
            real2string (yerr))
    end if
case (AN_PLOT)
    call analysis_iterator_get_data (iterator, &
        x = x, y = y, yerr = yerr, xerr = xerr, &
        index = point_index, n_total = n_points)
    if (has_writer) then
        call plot_writer_execute (writer, sandbox, file, &
            x, y, yerr, xerr, point_index, n_points)
    else if (columns) then

```

```

        call file_list_write (global%out_files, file, &
            real2string (x, OUT_REAL_FMT) // separator // &
            real2string (y, OUT_REAL_FMT) // separator // &
            real2string (yerr, OUT_REAL_FMT) // separator // &
            real2string (xerr, OUT_REAL_FMT))
    else
        call file_list_write (global%out_files, file, &
            real2string (x) // separator // &
            real2string (y) // separator // &
            real2string (yerr) // separator // &
            real2string (xerr))
    end if
end select
end subroutine write_data

end subroutine cmd_write_analysis_execute

```

Compile the histogram/plot writer: initialize special variables and compile the parse tree.

*{CCC Commands: procedures}+≡*

```

subroutine histogram_writer_compile (writer, pn, local, global)
    type(command_list_t), intent(out) :: writer
    type(parse_node_t), intent(in) :: pn
    type(rt_data_t), intent(inout), target :: local
    type(rt_data_t), intent(in), target :: global
    call rt_data_local_init (local, global)
    call var_list_append_real (local%var_list, &
        var_str ("bin_center"), intrinsic=.true.)
    call var_list_append_real (local%var_list, &
        var_str ("bin_width"), intrinsic=.true.)
    call var_list_append_real (local%var_list, &
        var_str ("bin_sum"), intrinsic=.true.)
    call var_list_append_real (local%var_list, &
        var_str ("bin_error"), intrinsic=.true.)
    call var_list_append_real (local%var_list, &
        var_str ("bin_excess"), intrinsic=.true.)
    call var_list_append_int (local%var_list, &
        var_str ("bin_index"), intrinsic=.true.)
    call var_list_append_int (local%var_list, &
        var_str ("n_bins"), intrinsic=.true.)
    call var_list_append_string (local%var_list, &
        var_str ("out_file"), intrinsic=.true.)
    call command_list_compile (writer, pn, local)
    call rt_data_local_reset (local)
end subroutine histogram_writer_compile

subroutine plot_writer_compile (writer, pn, local, global)
    type(command_list_t), intent(out) :: writer
    type(parse_node_t), intent(in) :: pn
    type(rt_data_t), intent(inout), target :: local
    type(rt_data_t), intent(in), target :: global
    call rt_data_local_init (local, global)
    call var_list_append_real (local%var_list, &

```

```

        var_str ("point_x"), intrinsic=.true.)
call var_list_append_real (local%var_list, &
    var_str ("point_y"), intrinsic=.true.)
call var_list_append_real (local%var_list, &
    var_str ("point_yerr"), intrinsic=.true.)
call var_list_append_real (local%var_list, &
    var_str ("point_xerr"), intrinsic=.true.)
call var_list_append_int (local%var_list, &
    var_str ("point_index"), intrinsic=.true.)
call var_list_append_int (local%var_list, &
    var_str ("n_points"), intrinsic=.true.)
call var_list_append_string (local%var_list, &
    var_str ("out_file"), intrinsic=.true.)
call command_list_compile (writer, pn, local)
call rt_data_local_reset (local)
end subroutine plot_writer_compile

```

Execute the histogram/plot writer: set special variables and execute the command list.

*{CCC Commands: procedures}+≡*

```

subroutine histogram_writer_execute (writer, local, filename, &
    x, width, y, yerr, excess, bin_index, n_bins)
    type(command_list_t), intent(in) :: writer
    type(rt_data_t), intent(inout), target :: local
    type(string_t), intent(in) :: filename
    real(default), intent(in) :: x, width, y, yerr, excess
    integer, intent(in) :: bin_index, n_bins
    call var_list_set_real (local%var_list, &
        var_str ("bin_center"), x, is_known=.true.)
    call var_list_set_real (local%var_list, &
        var_str ("bin_width"), width, is_known=.true.)
    call var_list_set_real (local%var_list, &
        var_str ("bin_sum"), y, is_known=.true.)
    call var_list_set_real (local%var_list, &
        var_str ("bin_error"), yerr, is_known=.true.)
    call var_list_set_real (local%var_list, &
        var_str ("bin_excess"), excess, is_known=.true.)
    call var_list_set_int (local%var_list, &
        var_str ("bin_index"), bin_index, is_known=.true.)
    call var_list_set_int (local%var_list, &
        var_str ("n_bins"), n_bins, is_known=.true.)
    call var_list_set_string (local%var_list, &
        var_str ("out_file"), filename, is_known=.true.)
    call command_list_execute (writer, local)
end subroutine histogram_writer_execute

subroutine plot_writer_execute (writer, local, filename, &
    x, y, yerr, xerr, point_index, n_points)
    type(command_list_t), intent(in) :: writer
    type(rt_data_t), intent(inout), target :: local
    type(string_t), intent(in) :: filename
    real(default), intent(in) :: x, y, yerr, xerr
    integer, intent(in) :: point_index, n_points

```

```

call var_list_set_real (local%var_list, &
    var_str ("point_x"), x, is_known=.true.)
call var_list_set_real (local%var_list, &
    var_str ("point_y"), y, is_known=.true.)
call var_list_set_real (local%var_list, &
    var_str ("point_yerr"), yerr, is_known=.true.)
call var_list_set_real (local%var_list, &
    var_str ("point_xerr"), xerr, is_known=.true.)
call var_list_set_int (local%var_list, &
    var_str ("point_index"), point_index, is_known=.true.)
call var_list_set_int (local%var_list, &
    var_str ("n_points"), n_points, &
    is_known=.true.)
call var_list_set_string (local%var_list, &
    var_str ("out_file"), filename, is_known=.true.)
call command_list_execute (writer, local)
end subroutine plot_writer_execute

```

## Defining writer macros

The following definitions are for the `histogram_writer` and `plot_writer` special variables. As they don't do much and are useful only in conjunction with `write_analysis` anyway, we don't open a separate section for them.

*(CCC Commands: types)+≡*

```

type :: cmd_xxx_writer_t
    type(parse_node_t), pointer :: macro => null ()
    integer :: type = CMD_HISTOGRAM_WRITER
end type cmd_xxx_writer_t

```

*(CCC Commands: procedures)+≡*

```

subroutine cmd_xxx_writer_final (writer)
    type(cmd_xxx_writer_t), intent(inout) :: writer
    nullify (writer%macro)
    writer%type = CMD_HISTOGRAM_WRITER
end subroutine cmd_xxx_writer_final

subroutine cmd_xxx_writer_compile (writer, pn, global)
    type(cmd_xxx_writer_t), intent(inout), pointer :: writer
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in) :: global
    allocate (writer)
    select case (char (parse_node_get_rule_key (pn)))
        case ("cmd_histogram_writer")
            writer%type = CMD_HISTOGRAM_WRITER
        case ("cmd_plot_writer")
            writer%type = CMD_PLOT_WRITER
        case default
            call msg_bug ("cmd_xxx_writer_compile: invalid type")
    end select
    writer%macro => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_xxx_writer_compile

```

```

subroutine cmd_xxx_writer_execute (writer, global)
  type(cmd_xxx_writer_t), intent(in), target :: writer
  type(rt_data_t), intent(inout), target :: global
  if (.not. associated (writer%macro)) return
  select case (writer%type)
    case (CMD_HISTOGRAM_WRITER)
      global%pn_histogram_writer => writer%macro
    case (CMD_PLOT_WRITER)
      global%pn_plot_writer => writer%macro
    case default
      call msg_bug ("cmd_xxx_writer_execute: invalid type")
  end select
end subroutine cmd_xxx_writer_execute

```

## Compile analysis results

This command writes files in a form suitable for GAMELAN and executes the appropriate commands to compile them. It reuses the `cmd_write_analysis_t` container defined above.

```

(CCC Commands: procedures)+≡
subroutine cmd_compile_analysis_execute (write, global)
  type(cmd_write_analysis_t), intent(inout), target :: write
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: data_file, basename, extension, driver_file
  integer :: u_driver, i
  logical :: has_gmlcode, write_yerr, write_xerr
  call cmd_write_analysis_execute &
    (write, global, data_file, write_yerr, write_xerr)
  basename = data_file
  if (scan (".", basename) > 0) then
    call split (basename, extension, ".", back=.true.)
  else
    extension = ""
  end if
  driver_file = basename // ".tex"
  u_driver = free_unit ()
  open (unit=u_driver, file=char(driver_file), &
    action="write", status="replace")
  call analysis_write_driver &
    (data_file, write%tag, unit=u_driver)
  close (u_driver)
  if (size (write%tag) == 0) then
    has_gmlcode = analysis_has_plots ()
  else
    has_gmlcode = analysis_has_plots (write%tag)
  end if
  call msg_message ("Compiling analysis results display in '" &
    // char (driver_file) // "'")
  call analysis_compile_tex (basename, has_gmlcode, global%os_data)
end subroutine cmd_compile_analysis_execute

```

## 20.2 User-controlled output to data files

### Open file (output)

Open a file for output.

*<CCC Commands: types>+≡*

```
type :: cmd_open_t
  private
  type(parse_node_t), pointer :: pn_sexpr => null ()
  logical :: reading = .false.
  logical :: writing = .false.
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_open_t
```

*<CCC Commands: procedures>+≡*

```
subroutine cmd_open_final (open)
  type(cmd_open_t), intent(inout) :: open
  if (associated (open%options)) then
    call command_list_final (open%options)
    deallocate (open%options)
  end if
end subroutine cmd_open_final

subroutine cmd_open_out_compile (open, pn, global)
  type(cmd_open_t), intent(out), pointer :: open
  type(parse_node_t), intent(in) :: pn
  type(rt_data_t), intent(in) :: global
  type(parse_node_t), pointer :: pn_arg, pn_opt
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  if (associated (pn_arg)) then
    pn_opt => parse_node_get_next_ptr (pn_arg)
  else
    pn_opt => null ()
  end if
  allocate (open)
  call rt_data_local_init (open%local, global)
  if (associated (pn_opt)) then
    allocate (open%options)
    call command_list_compile (open%options, pn_opt, open%local)
  end if
  open%pn_sexpr => pn_arg
  open%writing = .true.
  call rt_data_local_reset (open%local)
end subroutine cmd_open_out_compile

subroutine cmd_open_execute (open, global)
  type(cmd_open_t), intent(inout) :: open
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: name
  type(string_t) :: action, status, position
  call rt_data_link (open%local, global)
  if (associated (open%options)) then
    call command_list_execute (open%options, open%local)
```

```

end if
name = eval_string (open%pn_sexpr, open%local%var_list)
if (open%reading .and. open%writing) then
    action = "readwrite"
else if (open%reading) then
    action = "read"
else if (open%writing) then
    action = "write"
else
    call msg_bug ("Open command: neither read nor write specified.")
end if
status = "replace"
position = "asis"
call file_list_open (global%out_files, name, &
    char (action), char (status), char (position))
call rt_data_restore (global, open%local)
end subroutine cmd_open_execute

```

## Close file (output)

Close an output file.

*{CCC Commands: types}*+≡

```

type :: cmd_close_t
private
type(parse_node_t), pointer :: pn_sexpr => null ()
type(command_list_t), pointer :: options => null ()
type(rt_data_t) :: local
end type cmd_close_t

```

*{CCC Commands: procedures}*+≡

```

subroutine cmd_close_final (close)
type(cmd_close_t), intent(inout) :: close
if (associated (close%options)) then
    call command_list_final (close%options)
    deallocate (close%options)
end if
end subroutine cmd_close_final

subroutine cmd_close_out_compile (close, pn, global)
type(cmd_close_t), intent(out), pointer :: close
type(parse_node_t), intent(in) :: pn
type(rt_data_t), intent(in) :: global
type(parse_node_t), pointer :: pn_arg, pn_opt
pn_arg => parse_node_get_sub_ptr (pn, 2)
if (associated (pn_arg)) then
    pn_opt => parse_node_get_next_ptr (pn_arg)
else
    pn_opt => null ()
end if
allocate (close)
call rt_data_local_init (close%local, global)
if (associated (pn_opt)) then

```



```

        allocate (close%options)
        call command_list_compile (close%options, pn_opt, close%local)
    end if
    close%pn_sexpr => pn_arg
    call rt_data_local_reset (close%local)
end subroutine cmd_close_out_compile

subroutine cmd_close_execute (close, global)
    type(cmd_close_t), intent(inout) :: close
    type(rt_data_t), intent(inout), target :: global
    type(string_t) :: name
    call rt_data_link (close%local, global)
    if (associated (close%options)) then
        call command_list_execute (close%options, close%local)
    end if
    name = eval_string (close%pn_sexpr, close%local%var_list)
    call file_list_close (global%out_files, name)
    call rt_data_restore (global, close%local)
end subroutine cmd_close_execute

```

## 20.2.1 Print default-formatted values

*<CCC Commands: types>+≡*

```

type :: cmd_printd_t
    private
    type(parse_node_t), pointer :: pn_sexpr => null ()
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
end type cmd_printd_t

```

Finalize.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_printd_final (printd)
    type(cmd_printd_t), intent(inout) :: printd
    if (associated (printd%pn_sexpr)) then
        call parse_node_final (printd%pn_sexpr)
        deallocate (printd%pn_sexpr)
    end if
    if (associated (printd%options)) then
        call command_list_final (printd%options)
        deallocate (printd%options)
    end if
end subroutine cmd_printd_final

```

Compile. We create a fake parse node (subtree) with a `sprintd` command with identical arguments which can then be handled by the corresponding evaluation procedure.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_printd_compile (printd, pn, global)
    type(cmd_printd_t), pointer :: printd
    type(parse_node_t), intent(in), target :: pn

```

```

type(rt_data_t), intent(in), target :: global
type(parse_node_t), pointer :: pn_cmd, pn_arg, pn_opt
type(parse_node_t), pointer :: pn_sexpr, pn_sprind
pn_cmd => parse_node_get_sub_ptr (pn)
pn_opt => parse_node_get_next_ptr (pn_cmd)
pn_arg => parse_node_get_sub_ptr (pn_cmd)
allocate (prind)
call rt_data_local_init (prind%local, global)
if (associated (pn_opt)) then
    allocate (prind%options)
    call command_list_compile (prind%options, pn_opt, prind%local)
end if
call parse_node_create_branch (pn_sexpr, &
    syntax_get_rule_ptr (syntax_cmd_list, var_str ("sepr")))
call parse_node_create_branch (pn_sprind, &
    syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprind_fun")))
call parse_node_append_sub (pn_sexpr, pn_sprind)
call parse_node_append_sub (pn_sprind, pn_arg)
prind%pn_sexpr => pn_sexpr
call rt_data_local_reset (prind%local)
end subroutine cmd_prind_compile

```

Execute. Evaluate the string (pretending this is a `sprind` expression) and print it.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_prind_execute (prind, global)
    type(cmd_prind_t), intent(inout) :: prind
    type(rt_data_t), intent(inout), target :: global
    type(string_t) :: string, file
    logical :: advance
    call rt_data_link (prind%local, global)
    if (associated (prind%options)) then
        call command_list_execute (prind%options, prind%local)
    end if
    string = eval_string (prind%pn_sexpr, prind%local%var_list)
    file = var_list_get_sval (prind%local%var_list, var_str ("out_file"))
    advance = var_list_get_lval (prind%local%var_list, &
        var_str ("out_advance"))
    if (len (file) == 0) then
        call msg_result(char (string))
    else
        call file_list_write (global%out_files, file, string, advance)
    end if
    call rt_data_restore (global, prind%local)
end subroutine cmd_prind_execute

```

## Print custom-formatted values

*<CCC Commands: types>+≡*

```

type :: cmd_printf_t
private
type(parse_node_t), pointer :: pn_sexpr => null ()

```

```

        type(command_list_t), pointer :: options => null ()
        type(rt_data_t) :: local
    end type cmd_printf_t

```

Finalize.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_printf_final (printf)
    type(cmd_printf_t), intent(inout) :: printf
    if (associated (printf%pn_sexpr)) then
        call parse_node_final (printf%pn_sexpr)
        deallocate (printf%pn_sexpr)
    end if
    if (associated (printf%options)) then
        call command_list_final (printf%options)
        deallocate (printf%options)
    end if
end subroutine cmd_printf_final

```

Compile. We create a fake parse node (subtree) with a `sprintf` command with identical arguments which can then be handled by the corresponding evaluation procedure.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_printf_compile (printf, pn, global)
    type(cmd_printf_t), pointer :: printf
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_opt
    type(parse_node_t), pointer :: pn_sexpr, pn_sprintf
    pn_cmd => parse_node_get_sub_ptr (pn)
    pn_opt => parse_node_get_next_ptr (pn_cmd)
    pn_clause => parse_node_get_sub_ptr (pn_cmd)
    allocate (printf)
    call rt_data_local_init (printf%local, global)
    if (associated (pn_opt)) then
        allocate (printf%options)
        call command_list_compile (printf%options, pn_opt, printf%local)
    end if
    call parse_node_create_branch (pn_sexpr, &
        syntax_get_rule_ptr (syntax_cmd_list, var_str ("sexpr")))
    call parse_node_create_branch (pn_sprintf, &
        syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_fun")))
    call parse_node_append_sub (pn_sexpr, pn_sprintf)
    call parse_node_append_sub (pn_sprintf, pn_clause)
    printf%pn_sexpr => pn_sexpr
    call parse_node_final (pn_sprintf, recursive = .false.)
    call parse_node_final (pn_sexpr, recursive = .false.)
    call rt_data_local_reset (printf%local)
end subroutine cmd_printf_compile

```

Execute. Evaluate the string (pretending this is a `sprintf` expression) and print it.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_printf_execute (printf, global)
  type(cmd_printf_t), intent(inout) :: printf
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: string, file
  logical :: advance
  call rt_data_link (printf%local, global)
  if (associated (printf%options)) then
    call command_list_execute (printf%options, printf%local)
  end if
  string = eval_string (printf%pn_sexpr, printf%local%var_list)
  file = var_list_get_sval (printf%local%var_list, var_str ("%out_file"))
  advance = var_list_get_lval (printf%local%var_list, &
    var_str ("%out_advance"))
  if (len (file) == 0) then
    call msg_result (char (string))
  else
    call file_list_write (global%out_files, file, string, advance)
  end if
  call rt_data_restore (global, printf%local)
end subroutine cmd_printf_execute

```

## Record data

The expression syntax already contains a `record` keyword; this evaluates to a logical which is always true, but it has the side-effect of recording data into analysis objects. Here we define a command as an interface to this construct.

```

<CCC Commands: types>+≡
  type :: cmd_record_t
  private
  type(parse_node_t), pointer :: pn_lexpr => null ()
end type cmd_record_t

```

Finalizer for the eval trees.

```

<CCC Commands: procedures>+≡
  subroutine cmd_record_final (record)
    type(cmd_record_t), intent(inout) :: record
  end subroutine cmd_record_final

```

Compile. Record the record ID and initialize lower, upper bound and bin width.

```

<CCC Commands: procedures>+≡
  subroutine cmd_record_compile (record, pn, global)
    type(cmd_record_t), pointer :: record
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_lexpr, pn_lsinglet, pn_lterm, pn_record
    call parse_node_create_branch (pn_lexpr, &
      syntax_get_rule_ptr (syntax_cmd_list, var_str ("lexpr")))
    call parse_node_create_branch (pn_lsinglet, &
      syntax_get_rule_ptr (syntax_cmd_list, var_str ("lsinglet")))
    call parse_node_append_sub (pn_lexpr, pn_lsinglet)
    call parse_node_create_branch (pn_lterm, &

```

```

        syntax_get_rule_ptr (syntax_cmd_list, var_str ("lterm"))))
    call parse_node_append_sub (pn_lsinglet, pn_lterm)
    pn_record => parse_node_get_sub_ptr (pn)
    call parse_node_append_sub (pn_lterm, pn_record)
    allocate (record)
    record%pn_lexpr => pn_lexpr
end subroutine cmd_record_compile

```

Command execution.

```

<CCC Commands: procedures>+≡
subroutine cmd_record_execute (record, global)
    type(cmd_record_t), intent(inout), target :: record
    type(rt_data_t), intent(inout), target :: global
    logical :: lval
    lval = eval_log (record%pn_lexpr, global%var_list)
end subroutine cmd_record_execute

```

## Decay properties

This record holds decay properties for a particle. To be used for **unstable** and **stable** declarations, which contain an array of that type.

```

<CCC Commands: types>+≡
type :: decay_properties_t
    type(parse_node_t), pointer :: pn_pdg => null ()
    type(string_t) :: prt
    type(flavor_t) :: flv
    integer :: n_proc = 0
    type(string_t), dimension(:), allocatable :: process_id
    real(default), dimension(:), allocatable :: br
    logical :: invalid = .false.
end type decay_properties_t

```

The finalizer is generic:

```

<CCC Commands: procedures>+≡
subroutine decay_properties_final (decay)
    type(decay_properties_t), intent(inout) :: decay
end subroutine decay_properties_final

```

## Unstable particles

Mark a particle as unstable. For each unstable particle, we store a number of decay channels and compute their respective BRs.

```

<CCC Commands: types>+≡
type :: cmd_unstable_t
    private
    type(decay_properties_t), dimension(:), allocatable :: decay
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t) :: local
end type cmd_unstable_t

```

Delete the eval tree.

```

<CCC Commands: procedures>+≡
subroutine cmd_unstable_final (unstable)
  type(cmd_unstable_t), intent(inout) :: unstable
  integer :: d
  if (allocated (unstable%decay)) then
    do d = 1, size (unstable%decay)
      call decay_properties_final (unstable%decay(d))
    end do
    deallocate (unstable%decay)
  end if
  if (associated (unstable%options)) then
    call command_list_final (unstable%options)
    deallocate (unstable%options)
  end if
end subroutine cmd_unstable_final

```

Compile. Initiate an eval tree for the decaying particle and determine the decay channel process IDs.

```

<CCC Commands: procedures>+≡
subroutine cmd_unstable_compile (unstable, pn, global)
  type(cmd_unstable_t), pointer :: unstable
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_list, pn_decl
  type(parse_node_t), pointer :: pn_prt, pn_arg, pn_proc, pn_opt
  integer :: d, i
  pn_list => parse_node_get_sub_ptr (pn, 2)
  if (associated (pn_list)) then
    pn_opt => parse_node_get_next_ptr (pn_list)
  else
    pn_opt => null ()
  end if
  allocate (unstable)
  call rt_data_local_init (unstable%local, global)
  if (associated (pn_opt)) then
    allocate (unstable%options)
    call command_list_compile (unstable%options, pn_opt, unstable%local)
  end if
  allocate (unstable%decay (parse_node_get_n_sub (pn_list)))
  d = 0
  pn_decl => parse_node_get_sub_ptr (pn_list)
  do while (associated (pn_decl))
    d = d + 1
    pn_prt => parse_node_get_sub_ptr (pn_decl)
    pn_arg => parse_node_get_next_ptr (pn_prt)
    unstable%decay(d)%pn_pdg => pn_prt
    unstable%decay(d)%prt = "?"
    unstable%decay(d)%n_proc = parse_node_get_n_sub (pn_arg)
    allocate (unstable%decay(d)%process_id (unstable%decay(d)%n_proc))
    allocate (unstable%decay(d)%br (unstable%decay(d)%n_proc))
    pn_proc => parse_node_get_sub_ptr (pn_arg)
    do i = 1, unstable%decay(d)%n_proc

```

```

        unstable%decay(d)%process_id(i) = parse_node_get_string (pn_proc)
        pn_proc => parse_node_get_next_ptr (pn_proc)
    end do
    pn_decl => parse_node_get_next_ptr (pn_decl)
end do
call rt_data_local_reset (unstable%local)
end subroutine cmd_unstable_compile

```

Command execution. Evaluate the decaying particle and compute the BRs for the decays, using previous integration runs. For each decay channel, initialize event generation.

If the integral of some channel is not yet known, compute it.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_unstable_execute (unstable, global)
    type(cmd_unstable_t), intent(inout), target :: unstable
    type(rt_data_t), intent(inout), target :: global
    type(pdg_array_t) :: aval
    type(flavor_t), dimension(:), allocatable :: flv_tmp
    type(flavor_t), dimension(:), allocatable :: flv
    real(default), dimension(:), allocatable :: integral
    real(default) :: integral_sum
    type(process_t), pointer :: process
    type(string_t) :: process_id
    integer :: proc
    logical :: isotropic_decay, diagonal_decay
    type(particle_data_t), pointer :: prt_data
    type(decay_configuration_t), pointer :: decay_conf
    integer :: u, d
    u = logfile_unit ()
    call rt_data_link (unstable%local, global)
    if (associated (unstable%options)) then
        call command_list_execute (unstable%options, unstable%local)
    end if
    isotropic_decay = var_list_get_lval (unstable%local%var_list, &
        var_str ("?isotropic_decay"))
    diagonal_decay = var_list_get_lval (unstable%local%var_list, &
        var_str ("?diagonal_decay"))
    allocate (flv (size (unstable%decay)))
    do d = 1, size (unstable%decay)
        aval = eval_pdg_array (unstable%decay(d)%pn_pdg, unstable%local%var_list)
        call flavor_init (flv_tmp, aval, unstable%local%model)
        select case (size (flv_tmp))
        case (1); flv(d) = flv_tmp(1)
        case default
            call pdg_array_write (aval)
            call msg_fatal ("Unstable particle expression " &
                // "does not evaluate to a unique particle")
            return
        end select
        prt_data => model_get_particle_ptr &
            (unstable%local%model, flavor_get_pdg (flv(d)))
        if (associated (prt_data)) then
            if (flavor_is_polarized (flv(d))) then

```

```

        call msg_error ("particle '" // char (flavor_get_name (flv(d))) &
            // "' cannot be marked at unstable and polarized at the same " &
            // "time - skipping")
        unstable%decay(d)%invalid = .true.
        cycle
    end if
    if (flavor_is_antiparticle (flv(d))) then
        call particle_data_set (prt_data, &
            a_is_stable = .false., &
            a_decays_isotropically = isotropic_decay, &
            a_decays_diagonal = diagonal_decay)
    else
        call particle_data_set (prt_data, &
            p_is_stable = .false., &
            p_decays_isotropically = isotropic_decay, &
            p_decays_diagonal = diagonal_decay)
    end if
    unstable%decay(d)%invalid = .false.
else
    call msg_fatal ("Particle '" // char (unstable%decay(d)%prt) &
        // "' is not contained in model '" &
        // char (model_get_name (unstable%local%model)) // "')
    unstable%decay(d)%invalid = .true.
end if
end do
do d = 1, size (unstable%decay)
    if (unstable%decay(d)%invalid) cycle
    unstable%decay(d)%prt = flavor_get_name (flv(d))
    allocate (integral (unstable%decay(d)%n_proc))
    call integrate_missing_processes &
        (unstable%decay(d)%process_id, unstable%local, global%var_list, &
        no_beams=.true.)
    LOOP_PROC: do proc = 1, unstable%decay(d)%n_proc
        process_id = unstable%decay(d)%process_id(proc)
        process => process_store_get_process_ptr (process_id)
        if (associated (process)) then
            !
            !
            integral(proc) = var_list_get_rval (unstable%local%var_list, &
                var_str ("integral(" // process_id // ")")
            integral(proc) = process_get_integral (process)
            if (integral(proc) < 0) then
                call msg_fatal ("Integral of process '" &
                    // char (process_id) // "' is negative")
            end if
            call process_setup_event_generation (process, qn_mask_in = &
                new_quantum_numbers_mask (.false., .false., isotropic_decay, &
                mask_hd = diagonal_decay))
        else
            call msg_fatal ("Decay channel '" // char (process_id) &
                // "' is undefined")
        end if
    end do LOOP_PROC
    integral_sum = sum (integral)
    if (integral_sum /= 0) then
        unstable%decay(d)%br = integral / integral_sum
    end if
end do

```



```

else
  call msg_fatal ("Unstable particle: Computed total width vanishes")
  unstable%decay(d)%br = 0
end if
call decay_store_append_decay &
  (flv(d), unstable%local%model, &
   integral_sum, unstable%decay(d)%n_proc, &
   isotropic_decay, diagonal_decay, &
   decay_conf)
ASSIGN_DECAYS: do proc = 1, unstable%decay(d)%n_proc
  process => process_store_get_process_ptr &
    (unstable%decay(d)%process_id(proc))
  if (associated (process)) then
    call decay_configuration_set_channel &
      (decay_conf, proc, process, unstable%decay(d)%br(proc))
  end if
end do ASSIGN_DECAYS
deallocate (integral)
call msg_message ("Particle '" // char (unstable%decay(d)%prt) &
  // "' declared as unstable.")
call decay_configuration_write (decay_conf)
call decay_configuration_write (decay_conf, u)
end do
call decay_store_recheck_final_state (verbose = .true.)
call rt_data_restore (global, unstable%local)
end subroutine cmd_unstable_execute

```

## Stable particles

Mark a particle as stable. This is necessary only if it was previously marked as unstable.

```

<CCC Commands: types>+≡
  type :: cmd_stable_t
  private
  type(decay_properties_t), dimension(:), allocatable :: decay
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_stable_t

```

Delete the eval tree.

```

<CCC Commands: procedures>+≡
  subroutine cmd_stable_final (stable)
  type(cmd_stable_t), intent(inout) :: stable
  integer :: d
  if (allocated (stable%decay)) then
    do d = 1, size (stable%decay)
      call decay_properties_final (stable%decay(d))
    end do
    deallocate (stable%decay)
  end if
  if (associated (stable%options)) then
    call command_list_final (stable%options)
  end if
end subroutine cmd_stable_final

```

```

        deallocate (stable%options)
    end if
end subroutine cmd_stable_final

```

Compile. Initiate an eval tree for the particle code.

```

<CCC Commands: procedures>+≡
subroutine cmd_stable_compile (stable, pn, global)
    type(cmd_stable_t), pointer :: stable
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_list, pn_ptr, pn_opt
    integer :: d
    pn_list => parse_node_get_sub_ptr (pn, 2)
    pn_opt => parse_node_get_next_ptr (pn_list)
    allocate (stable)
    call rt_data_local_init (stable%local, global)
    if (associated (pn_opt)) then
        allocate (stable%options)
        call command_list_compile (stable%options, pn_opt, stable%local)
    end if
    allocate (stable%decay (parse_node_get_n_sub (pn_list)))
    d = 0
    pn_ptr => parse_node_get_sub_ptr (pn_list)
    do while (associated (pn_ptr))
        d = d + 1
        stable%decay(d)%pn_pdg => pn_ptr
        stable%decay(d)%prt = "?"
        pn_ptr => parse_node_get_next_ptr (pn_ptr)
    end do
    call rt_data_local_reset (stable%local)
end subroutine cmd_stable_compile

```

Command execution. Evaluate the particle type and undo any previous unstable declaration.

```

<CCC Commands: procedures>+≡
subroutine cmd_stable_execute (stable, global)
    type(cmd_stable_t), intent(inout), target :: stable
    type(rt_data_t), intent(inout), target :: global
    type(pdg_array_t) :: aval
    type(flavor_t), dimension(:), allocatable :: flv_tmp
    type(flavor_t) :: flv
    type(particle_data_t), pointer :: prt_data
    integer :: d
    call rt_data_link (stable%local, global)
    if (associated (stable%options)) then
        call command_list_execute (stable%options, stable%local)
    end if
    do d = 1, size (stable%decay)
        aval = eval_pdg_array (stable%decay(d)%pn_pdg, stable%local%var_list)
        call flavor_init (flv_tmp, aval, stable%local%model)
        select case (size (flv_tmp))
            case (1); flv = flv_tmp(1)
            case default

```

```

        call pdg_array_write (aval)
        call msg_fatal ("Stable particle expression " &
            // "does not evaluate to a unique particle")
        return
    end select
    stable%decay(d)%prt = flavor_get_name (flv)
    prt_data => &
        model_get_particle_ptr (stable%local%model, flavor_get_pdg (flv))
    if (associated (prt_data)) then
        if (flavor_is_antiparticle (flv)) then
            call particle_data_set (prt_data, a_is_stable=.true.)
            call particle_data_set (prt_data, a_decays_isotropically=.false.)
            call particle_data_set (prt_data, a_decays_diagonal=.false.)
        else
            call particle_data_set (prt_data, p_is_stable=.true.)
            call particle_data_set (prt_data, p_decays_isotropically=.false.)
            call particle_data_set (prt_data, p_decays_diagonal=.false.)
        end if
    else
        call msg_fatal ("Particle '" // char (stable%decay(d)%prt) &
            // "' is not contained in model '" &
            // char (model_get_name (stable%local%model)) // "'")
    end if
    call msg_message ("Particle '" // char (stable%decay(d)%prt) &
        // "' declared as stable.")
end do
call decay_store_recheck_final_state (verbose = .true.)
call rt_data_restore (global, stable%local)
end subroutine cmd_stable_execute

```

### (Un)polarized particles

Toggle flavors to be (un)polarized and make WHIZARD carry along the polarization. As the code is nearly duplicate between both case, we use a single data type and set of subroutines information.

*<CCC Commands: types>+≡*

```

type :: cmd_un_polarized_t
private
type(parse_node_p), dimension(:), allocatable :: pn_pdg
type(string_t), dimension(:), allocatable :: name
type(command_list_t), pointer :: options => null ()
type(rt_data_t) :: local
end type cmd_un_polarized_t

```

Compile. Initialize the data structure with the pdg expression trees.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_un_polarized_compile (polarized, pn, global)
type(cmd_un_polarized_t), pointer :: polarized
type(parse_node_t), intent(in), target :: pn
type(rt_data_t), intent(in), target :: global
type(parse_node_t), pointer :: pn_list, pn_opt, pn_prt
integer :: n
allocate (polarized)

```

```

pn_list => parse_node_get_sub_ptr (pn, 2)
call rt_data_local_init (polarized%local, global)
if (.not. associated (pn_list)) then
    allocate (polarized%pn_pdg (0))
    allocate (polarized%name (0))
    return
end if
pn_opt => parse_node_get_next_ptr (pn_list)
if (associated (pn_opt)) then
    allocate (polarized%options)
    call command_list_compile (polarized%options, pn_opt, polarized%local)
end if
n = parse_node_get_n_sub (pn_list)
allocate (polarized%pn_pdg(n), polarized%name(n))
polarized%name = "?"
pn_prt => parse_node_get_sub_ptr (pn_list)
n = 1
do while (associated (pn_prt))
    polarized%pn_pdg(n)%ptr => pn_prt
    pn_prt => parse_node_get_next_ptr (pn_prt)
    n = n + 1
end do
print *, "Processed " // int2char (n - 1) // " particle definitions"
call rt_data_local_reset (polarized%local)
end subroutine cmd_un_polarized_compile

```

Execute.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_un_polarized_execute (polarized, type, global)
    type(cmd_un_polarized_t), intent(inout) :: polarized
    integer, intent(in) :: type
    type(rt_data_t), target, intent(inout) :: global
    type(pdg_array_t) :: aval
    type(flavor_t), dimension(:), allocatable :: flv
    type(particle_data_t), pointer :: prt
    integer :: i, u
    logical :: anti
    call rt_data_link (polarized%local, global)
    if (associated (polarized%options)) &
        call command_list_execute (polarized%options, polarized%local)
    if (size (polarized%pn_pdg) == 0) return
    do i = 1, size (polarized%pn_pdg)
        aval = eval_pdg_array (polarized%pn_pdg(i)%ptr, polarized%local%var_list)
        call flavor_init (flv, aval, polarized%local%model)
        if (size (flv) /= 1) then
            call pdg_array_write (aval)
            u = output_unit ()
            if (u >= 0) write (u, "('')")
            call msg_error &
                ("'polarized' needs unique particles, ignoring argument")
            deallocate (flv)
            cycle
        end if
        polarized%name(i) = flavor_get_name (flv(1))
    end do
end subroutine cmd_un_polarized_execute

```

```

prt => model_get_particle_ptr (polarized%local%model, &
    flavor_get_pdg (flv(1)))
if (.not. associated (prt)) then
    call msg_error ("Model '" &
        // char (model_get_name (polarized%local%model)) &
        // "' does not contain particle '" &
        // char (polarized%name(i)) // "'")
    deallocate (flv)
    cycle
end if
anti = flavor_is_antiparticle (flv(1))
if (.not. particle_data_is_stable (prt, anti) .and. &
    type == CMD_POLARIZED) then
    call msg_error ("particle '" &
        // char (polarized%name(i)) // "' cannot be marked as unstable " &
        // "and polarized at the same time - skipping")
    deallocate (flv)
    cycle
end if
if (anti) then
    call particle_data_set (prt, a_polarized=(type == CMD_POLARIZED))
else
    call particle_data_set (prt, p_polarized=(type == CMD_POLARIZED))
end if
if (type == CMD_POLARIZED) then
    call msg_message ("Polarization of particle '" &
        // char (polarized%name(i)) // "' in model '" &
        // char (model_get_name (polarized%local%model)) &
        // "' will be retained.")
else
    call msg_message ("Polarization of particle '" &
        // char (polarized%name(i)) // "' in model '" &
        // char (model_get_name (polarized%local%model)) &
        // "' will be discarded.")
end if
deallocate (flv)
end do
call rt_data_restore (global, polarized%local)
end subroutine cmd_un_polarized_execute

```

Finalize.

*<CCC Commands: procedures>+≡*

```

subroutine cmd_un_polarized_final (polarized)
    type(cmd_un_polarized_t), intent(inout) :: polarized
    if (allocated (polarized%pn_pdg)) then
        deallocate (polarized%pn_pdg)
    end if
    if (allocated (polarized%name)) deallocate (polarized%name)
    if (associated (polarized%options)) then
        call command_list_final (polarized%options)
        deallocate (polarized%options)
    end if
end subroutine cmd_un_polarized_final

```

## Parameters: formats for event-sample output

Specify all event formats that are to be used for output files in the subsequent simulation run. (The raw format is on by default and can be turned off here.)

```
<Commands: types>+≡
    type, extends (command_t) :: cmd_sample_format_t
        private
        type(string_t), dimension(:), allocatable :: format
    contains
        <Commands: cmd sample format: TBP>
    end type cmd_sample_format_t
```

Finalizer.

```
<CCC Commands: procedures>+≡
    subroutine cmd_sample_format_final (sample)
        type(cmd_sample_format_t), intent(inout) :: sample
        if (allocated (sample%format)) then
            deallocate (sample%format)
        end if
    end subroutine cmd_sample_format_final
```

Output: here, everything is known.

```
<Commands: cmd sample format: TBP>≡
    procedure :: write => cmd_sample_format_write

<Commands: procedures>+≡
    subroutine cmd_sample_format_write (cmd, unit, indent)
        class(cmd_sample_format_t), intent(in) :: cmd
        integer, intent(in), optional :: unit, indent
        integer :: u, i
        u = output_unit (unit); if (u < 0) return
        call write_indent (u, indent)
        write (u, "(1x,A)", advance="no") "sample_format = "
        do i = 1, size (cmd%format)
            if (i > 1) write (u, "(A,1x)", advance="no") ", "
            write (u, "(A)", advance="no") char (cmd%format(i))
        end do
        write (u, "(A)")
    end subroutine cmd_sample_format_write
```

Compile. Initialize evaluation trees.

```
<Commands: cmd sample format: TBP>+≡
    procedure :: compile => cmd_sample_format_compile

<Commands: procedures>+≡
    subroutine cmd_sample_format_compile (cmd, global)
        class(cmd_sample_format_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(parse_node_t), pointer :: pn_arg
        type(parse_node_t), pointer :: pn_format
        integer :: i, n_format
        pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
        if (associated (pn_arg)) then
```

```

n_format = parse_node_get_n_sub (pn_arg)
allocate (cmd%format (n_format))
pn_format => parse_node_get_sub_ptr (pn_arg)
i = 0
do while (associated (pn_format))
  i = i + 1
  cmd%format(i) = parse_node_get_string (pn_format)
  pn_format => parse_node_get_next_ptr (pn_format)
end do
else
  allocate (cmd%format (0))
end if
end subroutine cmd_sample_format_compile

```

Execute. Transfer the list of format specifications to the corresponding array in the runtime data set.

```

<Commands: cmd sample format: TBP>+≡
  procedure :: execute => cmd_sample_format_execute

<Commands: procedures>+≡
  subroutine cmd_sample_format_execute (cmd, global)
    class(cmd_sample_format_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    if (allocated (global%sample_fmt)) deallocate (global%sample_fmt)
    allocate (global%sample_fmt (size (cmd%format)), source = cmd%format)
  end subroutine cmd_sample_format_execute

```

## The simulate command

This is the actual SINDARIN command.

```

<Commands: types>+≡
  type, extends (command_t) :: cmd_simulate_t
    ! not private anymore as required by the whizard-c-interface
    integer :: n_proc = 0
    type(string_t), dimension(:), allocatable :: process_id
  !
    type(command_list_t), pointer :: options => null ()
    type(rt_data_t), pointer :: local
  contains
    <Commands: cmd simulate: TBP>
  end type cmd_simulate_t

```

Finalizer.

```

<CCC Commands: procedures>+≡
  subroutine cmd_simulate_final (simulate)
    type(cmd_simulate_t), intent(inout) :: simulate
    if (associated (simulate%options)) then
      call command_list_final (simulate%options)
      deallocate (simulate%options)
    end if
  end subroutine cmd_simulate_final

```

Output: we know the process IDs.

*<Commands: cmd simulate: TBP>*≡

```
procedure :: write => cmd_simulate_write
```

*<Commands: procedures>*+≡

```
subroutine cmd_simulate_write (cmd, unit, indent)
  class(cmd_simulate_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "simulate ("
  do i = 1, cmd%n_proc
    if (i > 1) write (u, "(A,1x)", advance="no") ", "
    write (u, "(A)", advance="no") char (cmd%process_id(i))
  end do
  write (u, "(A)") ")"
end subroutine cmd_simulate_write
```

Compile. In contrast to WHIZARD 1 the confusing option to give the number of unweighted events for weighted events as if unweighting were to take place has been abandoned. (We both use `n_events` for weighted and unweighted events, the variable `n_calls` from WHIZARD 1 has been discarded.

*<Commands: cmd simulate: TBP>*+≡

```
procedure :: compile => cmd_simulate_compile
```

*<Commands: procedures>*+≡

```
subroutine cmd_simulate_compile (cmd, global)
  class(cmd_simulate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc, pn_opt
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_opt => parse_node_get_next_ptr (pn_proclist)
!   call rt_data_local_init (cmd%local, global)
  cmd%local => global
!   if (associated (pn_opt)) then
!     allocate (cmd%options)
!     call command_list_compile (cmd%options, pn_opt, cmd%local)
!   end if
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (cmd%process_id (cmd%n_proc))
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  do i = 1, cmd%n_proc
    cmd%process_id(i) = parse_node_get_string (pn_proc)
    pn_proc => parse_node_get_next_ptr (pn_proc)
  end do
!   call rt_data_local_reset (cmd%local)
end subroutine cmd_simulate_compile
```

Execute command: Simulate events. This is done via a `simulation_t` object and its associated methods.



Note: this temporary implementation uses only the `n_events` variable for counting. The full implementation should also accept the `luminosity` variable.

*(Commands: cmd simulate: TBP)+≡*

```
procedure :: execute => cmd_simulate_execute
```

*(Commands: procedures)+≡*

```
subroutine cmd_simulate_execute (cmd, global)
  class(cmd_simulate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
!   logical :: ok
!   integer :: i_evt
  integer :: n_events, n_fmt
  type(string_t) :: sample
  logical :: rebuild_events, read_raw, write_raw
  type(simulation_t), target :: sim
  type(string_t), dimension(:), allocatable :: sample_fmt
  type(event_stream_array_t) :: es_array
  type(event_sample_data_t) :: data
!   call rt_data_link (cmd%local, global)
!   if (associated (cmd%options)) then
!     call command_list_execute (cmd%options, cmd%local)
!   end if
  call sim%init (cmd%process_id, cmd%local)
  call sim%compute_weights ()
!   call openmp_set_num_threads_verbose &
!     (var_list_get_ival (global%var_list, "openmp_num_threads"))
  n_events = var_list_get_ival (cmd%local%var_list, var_str ("n_events"))
  sample = var_list_get_sval (cmd%local%var_list, var_str ("sample"))
  if (sample == "") sample = sim%get_default_sample_name ()
  rebuild_events = &
    var_list_get_lval (cmd%local%var_list, var_str ("rebuild_events"))
  read_raw = &
    var_list_get_lval (cmd%local%var_list, var_str ("read_raw")) &
    .and. .not. rebuild_events
  write_raw = &
    var_list_get_lval (cmd%local%var_list, var_str ("write_raw"))
  if (read_raw) then
    inquire (file = char (sample) // ".evx", exist = read_raw)
  end if
  if (allocated (cmd%local%sample_fmt)) then
    n_fmt = size (cmd%local%sample_fmt)
  else
    n_fmt = 0
  end if
  if (read_raw) then
    allocate (sample_fmt (n_fmt))
    if (n_fmt > 0) sample_fmt = cmd%local%sample_fmt
    call es_array%init (sample, &
      sample_fmt, sim%get_process_ptr (), cmd%local, &
      data = sim%get_data (), &
      input = var_str ("raw"), &
      allow_switch = write_raw)
    call sim%generate (n_events, es_array)
    call es_array%final ()
```

```

else if (write_raw) then
  allocate (sample_fmt (n_fmt + 1))
  if (n_fmt > 0) sample_fmt(:n_fmt) = cmd%local%sample_fmt
  sample_fmt(n_fmt+1) = var_str ("raw")
  call es_array%init (sample, &
    sample_fmt, sim%get_process_ptr (), cmd%local, &
    data = sim%get_data ())
  call sim%generate (n_events, es_array)
  call es_array%final ()
else if (allocated (cmd%local%sample_fmt)) then
  allocate (sample_fmt (n_fmt))
  if (n_fmt > 0) sample_fmt = cmd%local%sample_fmt
  call es_array%init (sample, &
    sample_fmt, sim%get_process_ptr (), cmd%local, &
    data = sim%get_data ())
  call sim%generate (n_events, es_array)
  call es_array%final ()
else
  call sim%generate (n_events)
end if
!      do while( simulation_get_i_evt(sim) .lt. simulation_get_n_events(sim))
!      call simulation_event (sim, cmd%local%rng, ok, global%os_data, verbose=.true.)
!      if (.not. ok) exit
!      end do
call sim%final ()
!      call rt_data_restore (global, cmd%local)
end subroutine cmd_simulate_execute

```

Execute command: Matching interface w/ PYTHIA

```

<CCC Commands: procedures>+≡
subroutine cmd_matching_execute (os_data)
  type(string_t) :: cmd_string
  type(os_data_t), intent(in) :: os_data
  call msg_message ("Starting MLM matching PYTHIA interface...")
  cmd_string = os_data%prefix // "/bin/interface"
  call os_system_call (cmd_string)
  call msg_message ("PYTHIA interface finished.")
end subroutine cmd_matching_execute

```

## The rescan command

This is the actual SINDARIN command.

```

<Commands: types>+≡
type, extends (command_t) :: cmd_rescan_t
!
!   private
!     type(parse_node_t), pointer :: pn_filename => null ()
!     integer :: n_proc = 0
!     type(string_t), dimension(:), allocatable :: process_id
!     type(command_list_t), pointer :: options => null ()
!     type(rt_data_t), pointer :: local
contains
<Commands: cmd rescan: TBP>

```

```
end type cmd_rescan_t
```

Finalizer.

```
<CCC Commands: procedures>+≡
subroutine cmd_rescan_final (rescan)
  type(cmd_rescan_t), intent(inout) :: rescan
  if (associated (rescan%options)) then
    call command_list_final (rescan%options)
    deallocate (rescan%options)
  end if
end subroutine cmd_rescan_final
```

Output: we know the process IDs.

```
<Commands: cmd rescan: TBP>≡
procedure :: write => cmd_rescan_write

<Commands: procedures>+≡
subroutine cmd_rescan_write (cmd, unit, indent)
  class(cmd_rescan_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit); if (u < 0) return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "rescan ("
  do i = 1, cmd%n_proc
    if (i > 1) write (u, "(A,1x)", advance="no") ", "
    write (u, "(A)", advance="no") char (cmd%process_id(i))
  end do
  write (u, "(A)") ")"
end subroutine cmd_rescan_write
```

Compile. The command takes a suffix argument, namely the file name of requested event file.

```
<Commands: cmd rescan: TBP>+≡
procedure :: compile => cmd_rescan_compile

<Commands: procedures>+≡
subroutine cmd_rescan_compile (cmd, global)
  class(cmd_rescan_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_filename, pn_proclist, pn_proc, pn_opt
  integer :: i
  pn_filename => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_proclist => parse_node_get_next_ptr (pn_filename)
  pn_opt => parse_node_get_next_ptr (pn_proclist)
!   call rt_data_local_init (cmd%local, global)
  cmd%local => global
!   if (associated (pn_opt)) then
!     allocate (cmd%options)
!     call command_list_compile (cmd%options, pn_opt, cmd%local)
!   end if
  cmd%pn_filename => pn_filename
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
```

```

allocate (cmd%process_id (cmd%n_proc))
pn_proc => parse_node_get_sub_ptr (pn_proclist)
do i = 1, cmd%n_proc
    cmd%process_id(i) = parse_node_get_string (pn_proc)
    pn_proc => parse_node_get_next_ptr (pn_proc)
end do
!    call rt_data_local_reset (cmd%local)
end subroutine cmd_rescan_compile

```

Execute command: Rescan events. This is done via a `simulation_t` object and its associated methods.

*(Commands: cmd rescan: TBP)+≡*

```

procedure :: execute => cmd_rescan_execute

```

*(Commands: procedures)+≡*

```

subroutine cmd_rescan_execute (cmd, global)
    class(cmd_rescan_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
!    logical :: ok
!    integer :: i_evt
    type(string_t) :: sample
    logical :: exist, write_raw, update_event, update_sqme, update_weight
    type(simulation_t), target :: sim
    type(event_sample_data_t) :: data
    type(string_t) :: input_sample
    integer :: n_fmt
    type(string_t), dimension(:), allocatable :: sample_fmt
    type(event_stream_array_t) :: es_array
    type(qcd_t) :: qcd
!    call rt_data_link (cmd%local, global)
!    if (associated (cmd%options)) then
!        call command_list_execute (cmd%options, cmd%local)
!    end if
    call sim%init (cmd%process_id, cmd%local)
    call sim%compute_weights ()
    input_sample = eval_string (cmd%pn_filename, cmd%local%var_list)
    sample = var_list_get_sval (cmd%local%var_list, var_str ("sample"))
    if (sample == "") sample = sim%get_default_sample_name ()
    write_raw = var_list_get_lval (cmd%local%var_list, var_str ("write_raw"))
    if (allocated (cmd%local%sample_fmt)) then
        n_fmt = size (cmd%local%sample_fmt)
    else
        n_fmt = 0
    end if
    if (write_raw) then
        if (sample == input_sample) then
            call msg_error ("Rescan: write_raw = true: " &
                // "suppressing raw event output (filename clashes with input)")
            allocate (sample_fmt (n_fmt))
            if (n_fmt > 0) sample_fmt = cmd%local%sample_fmt
        else
            allocate (sample_fmt (n_fmt + 1))
            if (n_fmt > 0) sample_fmt(:n_fmt) = cmd%local%sample_fmt
            sample_fmt(n_fmt+1) = var_str ("raw")
        end if
    end if
end subroutine cmd_rescan_execute

```

```

        end if
    else
        allocate (sample_fmt (n_fmt))
        if (n_fmt > 0) sample_fmt = cmd%local%sample_fmt
    end if
    update_event = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_event"))
    update_sqme = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_sqme"))
    update_weight = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_weight"))
    if (update_event .or. update_sqme) then
        call msg_message ("Recalculating observables")
        if (update_sqme) then
            call msg_message ("Recalculating squared matrix elements")
        end if
    end if
    inquire (file = char (input_sample) // ".evx", exist = exist)
    if (exist) then
        data = sim%get_data ()
        data%md5sum_cfg = ""
        call es_array%init (sample, &
            sample_fmt, sim%get_process_ptr (), cmd%local, data, &
            input = var_str ("raw"), input_sample = input_sample, &
            allow_switch = .false.)
        call dispatch_qcd (qcd, cmd%local)
        call sim%rescan (es_array, &
            update_event = update_event, &
            update_sqme = update_sqme, &
            update_weight = update_weight, &
            model = cmd%local%model, &
            helicity_selection = cmd%local%get_helicity_selection (), &
            qcd = qcd)
        call es_array%final ()
    else
        call msg_fatal ("Rescan: event file '" &
            // char (input_sample) // ".evx' not found")
    end if
    call sim%final ()
!    call rt_data_restore (global, rescan%local)
end subroutine cmd_rescan_execute

```

## 20.2.2 Parameters: random-number generator seed

Specify a new random-number generator seed and set it.

*(CCC Commands: types)*+≡

```

type :: cmd_seed_t
private
type(parse_node_t), pointer :: pn_expr => null ()
end type cmd_seed_t

```

Finalizer.

```
<CCC Commands: procedures>+≡  
  subroutine cmd_seed_final (seed)  
    type(cmd_seed_t), intent(inout) :: seed  
  end subroutine cmd_seed_final
```

Compile. Initialize evaluation trees.

```
<CCC Commands: procedures>+≡  
  subroutine cmd_seed_compile (seed, pn, global)  
    type(cmd_seed_t), pointer :: seed  
    type(parse_node_t), intent(in), target :: pn  
    type(rt_data_t), intent(in), target :: global  
    allocate (seed)  
    seed%pn_expr => parse_node_get_sub_ptr (pn, 3)  
  end subroutine cmd_seed_compile
```

Execute. Evaluate the trees and add the result to the iteration list in the runtime data set.

```
<CCC Commands: procedures>+≡  
  subroutine cmd_seed_execute (seed, global)  
    type(cmd_seed_t), intent(inout) :: seed  
    type(rt_data_t), intent(inout) :: global  
    integer :: seed_val  
    seed_val = eval_int (seed%pn_expr, global%var_list)  
    call set_rng_seed (global%rng, global%var_list, seed_val, verbose=.true.)  
  end subroutine cmd_seed_execute
```

```
<CCC Commands: procedures>+≡  
  subroutine set_rng_seed (rng, var_list, seed_val, verbose)  
    type(tao_random_state), intent(inout) :: rng  
    type(var_list_t), intent(inout), target :: var_list  
    integer, intent(in) :: seed_val  
    logical, intent(in) :: verbose  
    character(30) :: buffer  
    if (verbose) then  
      write (buffer, "(I0)") seed_val  
      call msg_message ("Setting seed for random-number generator to " &  
        // trim (buffer))  
    end if  
    call tao_random_seed (rng, seed_val)  
    call var_list_set_int (var_list, var_str ("seed_value"), &  
      seed_val, is_known=.true.)  
  end subroutine set_rng_seed
```

### Parameters: number of iterations

Specify number of iterations and number of calls for one integration pass.

```
<Commands: types>+≡  
  type, extends (command_t) :: cmd_iterations_t  
  private
```

```

integer :: n_pass = 0
type(parse_node_p), dimension(:), allocatable :: pn_expr_n_it
type(parse_node_p), dimension(:), allocatable :: pn_expr_n_calls
type(parse_node_p), dimension(:), allocatable :: pn_sexpr_adapt
contains
  <Commands: cmd iterations: TBP>
end type cmd_iterations_t

```

Finalizer.

```

<CCC Commands: cmd iterations: TBP>≡
  procedure :: final => cmd_iterations_final

<CCC Commands: procedures>+≡
  subroutine cmd_iterations_final (cmd)
    type(cmd_iterations_t), intent(inout) :: cmd
    integer :: i
    if (allocated (cmd%pn_expr_n_it)) then
      deallocate (cmd%pn_expr_n_it)
    end if
    if (allocated (cmd%pn_expr_n_calls)) then
      deallocate (cmd%pn_expr_n_calls)
    end if
  end subroutine cmd_iterations_final

```

Output. Display the number of passes, which is known after compilation.

```

<Commands: cmd iterations: TBP>≡
  procedure :: write => cmd_iterations_write

<Commands: procedures>+≡
  subroutine cmd_iterations_write (cmd, unit, indent)
    class(cmd_iterations_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit); if (u < 0) return
    call write_indent (u, indent)
    select case (cmd%n_pass)
    case (0)
      write (u, "(1x,A)") "iterations: [empty]"
    case (1)
      write (u, "(1x,A,I0,A)") "iterations: ", cmd%n_pass, " pass"
    case default
      write (u, "(1x,A,I0,A)") "iterations: ", cmd%n_pass, " passes"
    end select
  end subroutine cmd_iterations_write

```

Compile. Initialize evaluation trees.

```

<Commands: cmd iterations: TBP>+≡
  procedure :: compile => cmd_iterations_compile

<Commands: procedures>+≡
  subroutine cmd_iterations_compile (cmd, global)
    class(cmd_iterations_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_n_it, pn_n_calls, pn_adapt

```

```

type(parse_node_t), pointer :: pn_it_spec, pn_calls_spec, pn_adapt_spec
integer :: i
pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
if (associated (pn_arg)) then
  cmd%n_pass = parse_node_get_n_sub (pn_arg)
  allocate (cmd%pn_expr_n_it (cmd%n_pass))
  allocate (cmd%pn_expr_n_calls (cmd%n_pass))
  allocate (cmd%pn_sexpr_adapt (cmd%n_pass))
  pn_it_spec => parse_node_get_sub_ptr (pn_arg)
  i = 1
  do while (associated (pn_it_spec))
    pn_n_it => parse_node_get_sub_ptr (pn_it_spec)
    pn_calls_spec => parse_node_get_next_ptr (pn_n_it)
    pn_n_calls => parse_node_get_sub_ptr (pn_calls_spec, 2)
    pn_adapt_spec => parse_node_get_next_ptr (pn_calls_spec)
    if (associated (pn_adapt_spec)) then
      pn_adapt => parse_node_get_sub_ptr (pn_adapt_spec, 2)
    else
      pn_adapt => null ()
    end if
    cmd%pn_expr_n_it(i)%ptr => pn_n_it
    cmd%pn_expr_n_calls(i)%ptr => pn_n_calls
    cmd%pn_sexpr_adapt(i)%ptr => pn_adapt
    i = i + 1
    pn_it_spec => parse_node_get_next_ptr (pn_it_spec)
  end do
else
  allocate (cmd%pn_expr_n_it (0))
  allocate (cmd%pn_expr_n_calls (0))
end if
end subroutine cmd_iterations_compile

```

Execute. Evaluate the trees and transfer the results to the iteration list in the runtime data set.

*(Commands: cmd iterations: TBP)+≡*

```

procedure :: execute => cmd_iterations_execute

```

*(Commands: procedures)+≡*

```

subroutine cmd_iterations_execute (cmd, global)
class(cmd_iterations_t), intent(inout) :: cmd
type(rt_data_t), intent(inout), target :: global
integer, dimension(cmd%n_pass) :: n_it, n_calls
logical, dimension(cmd%n_pass) :: custom_adapt
type(string_t), dimension(cmd%n_pass) :: adapt_code
integer :: i
do i = 1, cmd%n_pass
  n_it(i) = eval_int (cmd%pn_expr_n_it(i)%ptr, global%var_list)
  n_calls(i) = &
    eval_int (cmd%pn_expr_n_calls(i)%ptr, global%var_list)
  if (associated (cmd%pn_sexpr_adapt(i)%ptr)) then
    adapt_code(i) = &
      eval_string (cmd%pn_sexpr_adapt(i)%ptr, &
        global%var_list, is_known = custom_adapt(i))
  else

```



```

        custom_adapt(i) = .false.
    end if
end do
    call global%it_list%init (n_it, n_calls, custom_adapt, adapt_code)
end subroutine cmd_iterations_execute

```

## Scan over parameters and other objects

```

(Commands: parameters)≡
    integer, parameter :: STEP_NONE = 0
    integer, parameter :: STEP_ADD = 1
    integer, parameter :: STEP_SUB = 2
    integer, parameter :: STEP_MUL = 3
    integer, parameter :: STEP_DIV = 4
    integer, parameter :: STEP_COMP_ADD = 11
    integer, parameter :: STEP_COMP_MUL = 13

(CCC Commands: types)+=≡
    type :: cmd_scan_t
    private
    integer :: var_type = V_NONE
    type(string_t) :: var_name
    logical :: allow_steps = .false.
    integer :: n_arg = 0
    type(command_list_t), dimension(:), pointer :: cmd_var => null ()
    logical, dimension(:), allocatable :: has_range
    type(parse_node_p), dimension(:), allocatable :: pn_beg_expr
    type(parse_node_p), dimension(:), allocatable :: pn_end_expr
    integer, dimension(:), allocatable :: step_type
    type(parse_node_p), dimension(:), allocatable :: pn_step_expr
    type(command_list_t), pointer :: body => null ()
    type(rt_data_t) :: local
end type cmd_scan_t

```

Finalizer.

```

(CCC Commands: procedures)+=≡
    recursive subroutine cmd_scan_final (loop)
    type(cmd_scan_t), intent(inout) :: loop
    integer :: i
    if (associated (loop%cmd_var)) then
        do i = 1, size (loop%cmd_var)
            call command_list_final (loop%cmd_var(i))
        end do
        deallocate (loop%cmd_var)
    end if
    if (allocated (loop%has_range)) deallocate (loop%has_range)
    if (allocated (loop%pn_beg_expr)) then
        deallocate (loop%pn_beg_expr)
    end if
    if (allocated (loop%pn_end_expr)) then
        deallocate (loop%pn_end_expr)
    end if
    if (allocated (loop%step_type)) deallocate (loop%step_type)

```

```

    if (allocated (loop%pn_step_expr)) then
        deallocate (loop%pn_step_expr)
    end if
    if (associated (loop%body)) then
        call command_list_final (loop%body)
        deallocate (loop%body)
    end if
end subroutine cmd_scan_final

```

Compile the loop, including the list of step specifications.

*(CCC Commands: procedures)*+≡

```

recursive subroutine cmd_scan_compile (loop, pn, global)
    type(cmd_scan_t), pointer :: loop
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_spec, pn_cmd, pn_list, pn_body
    integer :: i
    type(parse_tree_t) :: parse_tree
    type(parse_node_t), pointer :: pn_root, pn_decl, pn_init_arg, pn_arg
    type(parse_node_t), pointer :: pn_expr, pn_range, pn_range_expr
    type(parse_node_t), pointer :: pn_step, pn_step_op, pn_step_expr
    type(parse_node_t), pointer :: pn_var, pn_name
    type(string_t) :: key, str_init
    type(lexer_t) :: lexer
    type(stream_t), target :: stream
    logical :: declaration, new
    pn_spec => parse_node_get_sub_ptr (pn)
    pn_cmd => parse_node_get_sub_ptr (pn_spec, 2)
    pn_var => pn_cmd
    allocate (loop)
    call rt_data_local_init (loop%local, global)
    if (associated (pn_cmd)) then
        key = parse_node_get_rule_key (pn_cmd)
    else
        loop%var_type = V_NONE
        loop%n_arg = 0
        return
    end if
    declaration = .false.
    select case (char (key))
    case ("cmd_model_list")
        str_init = 'model = ""'
    case ("cmd_library_list")
        str_init = 'library = ""'
    case ("cmd_seed_list")
        loop%var_type = V_INT
        loop%var_name = "seed"
        str_init = 'seed = 0'
        loop%allow_steps = .true.
    case ("cmd_cuts_list")
        str_init = 'cuts = true'
    case ("cmd_scale_list")
        str_init = 'scale = 0'
    case ("cmd_fac_scale_list")

```

```

        str_init = 'factorization_scale = 0'
case ("cmd_ren_scale_list")
    str_init = 'renormalization_scale = 0'
case ("cmd_weight_list")
    str_init = 'weight = 0'
case ("cmd_selection_list")
    str_init = 'selection = true'
case ("cmd_reweight_list")
    str_init = 'reweight = 0'
case ("cmd_analysis_list")
    str_init = 'analysis = true'
case default
    new = .false.
    select case (char (key))
    case ("log_decl_list")
        loop%var_type = V_LOG
        pn_var => parse_node_get_sub_ptr (pn_cmd, 2)
        pn_name => parse_node_get_sub_ptr (pn_var, 2)
        loop%var_name = '?' // parse_node_get_string (pn_name)
        str_init = 'logical ' // loop%var_name // ' = true'
        declaration = .true.
        new = .true.
    case ("log_list")
        loop%var_type = V_LOG
        pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
        loop%var_name = '?' // parse_node_get_string (pn_name)
        str_init = loop%var_name // ' = true'
    case ("int_list")
        loop%var_type = V_INT
        pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
        loop%var_name = parse_node_get_string (pn_name)
        str_init = 'int ' // loop%var_name // ' = 0'
        loop%allow_steps = .true.
        new = .true.
    case ("real_list")
        loop%var_type = V_REAL
        pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
        loop%var_name = parse_node_get_string (pn_name)
        str_init = 'real ' // loop%var_name // ' = 0'
        loop%allow_steps = .true.
        new = .true.
    case ("num_list")
        loop%var_type = V_REAL
        pn_name => parse_node_get_sub_ptr (pn_cmd)
        loop%var_name = parse_node_get_string (pn_name)
        str_init = loop%var_name // ' = 0'
        loop%allow_steps = .true.
    case ("string_decl_list")
        loop%var_type = V_STR
        pn_var => parse_node_get_sub_ptr (pn_cmd, 2)
        pn_name => parse_node_get_sub_ptr (pn_var, 2)
        loop%var_name = '$' // parse_node_get_string (pn_name)
        str_init = 'string ' // loop%var_name // ' = ""'
        declaration = .true.

```

```

        new = .true.
    case ("string_list")
        loop%var_type = V_STR
        pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
        loop%var_name = '$' // parse_node_get_string (pn_name)
        str_init = loop%var_name // ' = ""'
    case ("alias_list")
        loop%var_type = V_PDG
        pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
        loop%var_name = parse_node_get_string (pn_name)
        str_init = 'alias ' // loop%var_name // ' = PDG(0)'
        new = .true.
    case default
        call parse_node_mismatch &
            ("scan: model|library|seed|cuts|weight|" // &
             "scale|factorization_scale|renormalization_scale|analysis|" &
             // "variable", pn_cmd)
    end select
    call var_list_check_user_var &
        (global%var_list, loop%var_name, loop%var_type, new)
    if (loop%var_type == V_NONE) then
        call msg_fatal ("Invalid scan variable declaration")
        loop%n_arg = 0
        return
    end if
end select
if (associated (pn_var)) then
    pn_list => parse_node_get_last_sub_ptr (pn_var)
    if (associated (pn_list)) then
        loop%n_arg = parse_node_get_n_sub (pn_list)
        allocate (loop%cmd_var (loop%n_arg))
        call lexer_init_cmd_list (lexer)
        allocate (loop%has_range (loop%n_arg)); loop%has_range = .false.
        if (loop%allow_steps) then
            allocate (loop%pn_beg_expr (loop%n_arg))
            allocate (loop%pn_end_expr (loop%n_arg))
            allocate (loop%step_type (loop%n_arg)); loop%step_type = STEP_NONE
            allocate (loop%pn_step_expr (loop%n_arg))
        end if
        call stream_init (stream, str_init)
        call lexer_assign_stream (lexer, stream)
        call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
        pn_root => parse_tree_get_root_ptr (parse_tree)
        if (declaration) then
            pn_decl => parse_node_get_sub_ptr (pn_root)
            pn_cmd => parse_node_get_sub_ptr (pn_decl, 2)
        else
            pn_cmd => parse_node_get_sub_ptr (pn_root)
        end if
        pn_init_arg => parse_node_get_last_sub_ptr (pn_cmd)
        i = 0
        pn_arg => parse_node_get_sub_ptr (pn_list)
        do while (associated (pn_arg))
            i = i + 1

```

```

select case (char (parse_node_get_rule_key (pn_arg)))
case ("num_steps")
  pn_expr => parse_node_get_sub_ptr (pn_arg)
  pn_range => parse_node_get_next_ptr (pn_expr)
  if (associated (pn_range)) then
    loop%has_range(i) = .true.
    pn_range_expr => parse_node_get_sub_ptr (pn_range, 2)
    loop%pn_beg_expr(i)%ptr => pn_expr
    loop%pn_end_expr(i)%ptr => pn_range_expr
    pn_step => parse_node_get_next_ptr (pn_range_expr)
    if (associated (pn_step)) then
      pn_step_op => parse_node_get_sub_ptr (pn_step)
      select case (char (parse_node_get_key (pn_step_op)))
      case ("/+"); loop%step_type(i) = STEP_ADD
      case ("/-"); loop%step_type(i) = STEP_SUB
      case ("/*"); loop%step_type(i) = STEP_MUL
      case ("//"); loop%step_type(i) = STEP_DIV
      case ("++"); loop%step_type(i) = STEP_COMP_ADD
      case ("**"); loop%step_type(i) = STEP_COMP_MUL
      end select
      pn_step_expr => parse_node_get_next_ptr (pn_step_op)
      loop%pn_step_expr(i)%ptr => pn_step_expr
    end if
  end if
case default
  pn_expr => pn_arg
end select
call parse_node_replace_last_sub (pn_cmd, pn_expr)
call command_list_compile (loop%cmd_var(i), pn_root, loop%local)
pn_arg => parse_node_get_next_ptr (pn_arg)
end do
call parse_node_replace_last_sub (pn_cmd, pn_init_arg)
call parse_tree_final (parse_tree)
call stream_final (stream)
call lexer_final (lexer)
end if
pn_body => parse_node_get_next_ptr (pn_spec)
if (associated (pn_body)) then
  allocate (loop%body)
  call command_list_compile (loop%body, pn_body, loop%local)
end if
else
  loop%n_arg = 0
end if
call rt_data_local_reset (loop%local)
end subroutine cmd_scan_compile

```

Execute the loop for all values in the step list.

```

<CCC Commands: procedures>+≡
recursive subroutine cmd_scan_execute (loop, global)
  type(cmd_scan_t), intent(inout), target :: loop
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: model_name
  logical :: is_seed

```

```

integer :: i, j
integer :: i1, i2, istep, ival, n_steps
real(default) :: r1, r2, rstep, rval, rlog, r1log, r2log
logical :: is_known
type(var_entry_t), pointer :: var
type(var_list_t), pointer :: model_vars
call rt_data_link (loop%local, global)
if (associated (loop%local%model)) then
    model_name = model_get_name (loop%local%model)
    model_vars => model_get_var_list_ptr (loop%local%model)
else
    model_vars => null ()
end if
LOOP_LIST: do i = 1, loop%n_arg
    call command_list_execute (loop%cmd_var(i), loop%local)
    if (.not. loop%has_range(i)) then
        if (associated (loop%body)) then
            call command_list_execute (loop%body, loop%local)
            if (loop%local%quit) then
                global%quit_code = loop%local%quit_code
                global%quit = .true.
                return
            end if
        end if
    else
        call eval_numeric (loop%pn_beg_expr(i)%ptr, loop%local%var_list, &
            is_known=is_known, ival=i1, rval=r1)
        if (.not. is_known) then
            call msg_error ("Scan: undefined lower bound, skipping")
            cycle LOOP_LIST
        end if
        call eval_numeric (loop%pn_end_expr(i)%ptr, loop%local%var_list, &
            is_known=is_known, ival=i2, rval=r2)
        if (.not. is_known) then
            call msg_error ("Scan: undefined upper bound, skipping")
            cycle LOOP_LIST
        end if
        select case (loop%step_type(i))
        case (STEP_NONE)
            rstep = 1
            istep = 1
        case default
            call eval_numeric (loop%pn_step_expr(i)%ptr, loop%local%var_list, &
                is_known=is_known, ival=istep, rval=rstep)
            if (.not. is_known) then
                call msg_error ("Scan: undefined step size, skipping")
                cycle LOOP_LIST
            end if
        end select
        if (bounds_check_fails (loop%step_type(i), loop%var_type)) &
            cycle LOOP_LIST
        if (loop%var_name == "seed") then
            is_seed = .true.
        else

```

```

is_seed = .false.
var => var_list_get_var_ptr (loop%local%var_list, loop%var_name)
if (var_entry_get_type (var) /= loop%var_type) then
    call msg_fatal ("Type mismatch for loop variable '" &
        // char (loop%var_name) // "; skipping")
    exit LOOP_LIST
end if
end if
select case (loop%var_type)
case (V_REAL)
    n_steps = -1
    select case (loop%step_type(i))
    case (STEP_NONE, STEP_ADD)
        if (rstep /= 0) n_steps = nint ((r2 - r1) / rstep)
    case (STEP_SUB)
        if (rstep /= 0) n_steps = nint (- (r2 - r1) / rstep)
    case (STEP_COMP_ADD)
        if (istep /= 0) n_steps = istep
    case (STEP_MUL, STEP_DIV, STEP_COMP_MUL)
        if (r1 * r2 > 0) then
            if (rstep > 0 .and. rstep /= 1) then
                r1log = log (abs (r1))
                r2log = log (abs (r2))
                select case (loop%step_type(i))
                case (STEP_MUL)
                    n_steps = nint (log (r2 / r1) / log (rstep))
                case (STEP_DIV)
                    n_steps = nint (- log (r2 / r1) / log (rstep))
                case (STEP_COMP_MUL)
                    n_steps = istep
                end select
            end if
        end if
    end select
end select
if (n_steps == 0) n_steps = -1
select case (loop%step_type(i))
case (STEP_NONE, STEP_ADD, STEP_COMP_ADD, STEP_SUB)
    do j = 0, n_steps
        rval = (j * r2 + (n_steps-j) * r1) / n_steps
        call set_real (rval, j/=0)
        if (associated (loop%body)) then
            call command_list_execute (loop%body, loop%local)
            if (loop%local%quit) then
                global%quit_code = loop%local%quit_code
                global%quit = .true.
                return
            end if
        end if
    end do
case (STEP_MUL, STEP_COMP_MUL, STEP_DIV)
    do j = 0, n_steps
        rlog = (j * r2log + (n_steps-j) * r1log) / n_steps
        rval = sign (exp (rlog), r1)
        call set_real (rval, j/=0)

```

```

        if (associated (loop%body)) then
            call command_list_execute (loop%body, loop%local)
            if (loop%local%quit) then
                global%quit_code = loop%local%quit_code
                global%quit = .true.
                return
            end if
        end if
    end do
end select
case (V_INT)
    select case (loop%step_type(i))
    case (STEP_SUB); istep = - istep
    end select
    select case (loop%step_type(i))
    case (STEP_NONE, STEP_ADD, STEP_SUB)
        do j = i1, i2, istep
            call set_int (j, j/=i1, is_seed)
            if (associated (loop%body)) then
                call command_list_execute (loop%body, loop%local)
                if (loop%local%quit) then
                    global%quit_code = loop%local%quit_code
                    global%quit = .true.
                    return
                end if
            end if
        end do
    case (STEP_MUL, STEP_DIV)
        call msg_error ("Skipping scan: " &
            // "Multiplicative steps not allowed " &
            // "for integer scan variable")
    case (STEP_COMP_ADD, STEP_COMP_MUL)
        call msg_error ("Skipping scan: " &
            // "Range division not allowed for integer scan variable")
    end select
end select
end if
end do LOOP_LIST
call rt_data_restore (global, loop%local)
contains
subroutine set_real (rval, verbose)
    real(default), intent(in) :: rval
    logical, intent(in) :: verbose
    call var_entry_set_real (var, rval, is_known=.true., verbose=verbose, &
        model_name=model_name)
    if (var_entry_is_copy (var)) then
        call model_parameters_update (loop%local%model)
        call var_list_synchronize (loop%local%var_list, model_vars)
    end if
end subroutine set_real
subroutine set_int (ival, verbose, is_seed)
    integer, intent(in) :: ival
    logical, intent(in) :: verbose, is_seed
    if (is_seed) then

```



```

        call set_rng_seed (loop%local%rng, loop%local%var_list, ival, &
                           verbose=verbose)
    else
        call var_entry_set_int (var, ival, is_known=.true., verbose=verbose, &
                               model_name=model_name)
        if (var_entry_is_copy (var)) then
            call model_parameters_update (loop%local%model)
            call var_list_synchronize (loop%local%var_list, model_vars)
        end if
    end if
end subroutine set_int
function bounds_check_fails (step_type, var_type) result (fails)
    logical :: fails
    integer, intent(in) :: step_type, var_type
    fails = .false.
    select case (var_type)
    case (V_REAL)
        if (rstep == 0) then
            call msg_error ("Scan: step size equals zero, skipping")
            fails = .true.; return
        end if
        select case (step_type)
        case (STEP_MUL, STEP_DIV)
            if (rstep < 0) then
                call msg_error ("Scan: multiplicative step < 0, skipping")
                fails = .true.; return
            else if (rstep == 1) then
                call msg_error ("Scan: multiplicative step = 1, skipping")
                fails = .true.; return
            else if (r1 == 0 .or. r2 == 0) then
                call msg_error ("Scan: " &
                               // "boundary for multiplicative step is zero, skipping")
                fails = .true.; return
            end if
        end select
    case (V_INT)
        if (istep == 0) then
            call msg_error ("Scan: step size equals zero, skipping")
            fails = .true.; return
        end if
        select case (step_type)
        case (STEP_MUL, STEP_DIV)
            if (istep < 0) then
                call msg_error ("Scan: multiplicative step < 0, skipping")
                fails = .true.; return
            else if (istep == 1) then
                call msg_error ("Scan: multiplicative step = 1, skipping")
                fails = .true.; return
            else if (i1 == 0 .or. i2 == 0) then
                call msg_error ("Scan: " &
                               // "boundary for multiplicative step is zero, skipping")
                fails = .true.; return
            end if
        end select
    end select
end function

```

```

        end select
    end function bounds_check_fails
end subroutine cmd_scan_execute

```

## Conditionals

Conditionals are implemented as a list that is compiled and evaluated recursively; this allows for a straightforward representation of **else if** constructs. A `cmd_if_t` object can hold either an `else_if` clause which is another object of this type, or an `else_body`, but not both.

If- or else-bodies are no scoping units, so all data remain global and no copy-in copy-out is needed.

*(CCC Commands: types)+≡*

```

type :: cmd_if_t
    private
    type(parse_node_t), pointer :: pn_if_lexpr => null ()
    type(command_list_t), pointer :: if_body => null ()
    type(cmd_if_t), dimension(:), pointer :: elsif_cond => null ()
    type(command_list_t), pointer :: else_body => null ()
end type cmd_if_t

```

Finalizer.

*(CCC Commands: procedures)+≡*

```

recursive subroutine cmd_if_final (cond)
    type(cmd_if_t), intent(inout) :: cond
    integer :: i
    if (associated (cond%if_body)) then
        call command_list_final (cond%if_body)
        deallocate (cond%if_body)
    end if
    if (associated (cond%elsif_cond)) then
        do i = 1, size (cond%elsif_cond)
            call cmd_if_final (cond%elsif_cond(i))
        end do
        deallocate (cond%elsif_cond)
    end if
    if (associated (cond%else_body)) then
        call command_list_final (cond%else_body)
        deallocate (cond%else_body)
    end if
end subroutine cmd_if_final

```

Compile the conditional.

*(CCC Commands: procedures)+≡*

```

recursive subroutine cmd_if_compile (cond, pn, global)
    type(cmd_if_t), pointer :: cond
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_lexpr, pn_body
    type(parse_node_t), pointer :: pn_elsif_clauses, pn_cmd_elsif
    type(parse_node_t), pointer :: pn_else_clause, pn_cmd_else

```

```

integer :: i, n_elseif
allocate (cond)
pn_lexpr => parse_node_get_sub_ptr (pn, 2)
cond%pn_if_lexpr => pn_lexpr
pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
select case (char (parse_node_get_rule_key (pn_body)))
case ("command_list")
    allocate (cond%if_body)
    call command_list_compile (cond%if_body, pn_body, global)
    pn_elseif_clauses => parse_node_get_next_ptr (pn_body)
case default
    pn_elseif_clauses => pn_body
end select
select case (char (parse_node_get_rule_key (pn_elseif_clauses)))
case ("elseif_clauses")
    n_elseif = parse_node_get_n_sub (pn_elseif_clauses)
    allocate (cond%elseif_cond (n_elseif))
    pn_cmd_elseif => parse_node_get_sub_ptr (pn_elseif_clauses)
    do i = 1, n_elseif
        pn_lexpr => parse_node_get_sub_ptr (pn_cmd_elseif, 2)
        cond%elseif_cond(i)%pn_if_lexpr => pn_lexpr
        pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
        if (associated (pn_body)) then
            allocate (cond%elseif_cond(i)%if_body)
            call command_list_compile &
                (cond%elseif_cond(i)%if_body, pn_body, global)
        end if
        pn_cmd_elseif => parse_node_get_next_ptr (pn_cmd_elseif)
    end do
    pn_else_clause => parse_node_get_next_ptr (pn_elseif_clauses)
case default
    pn_else_clause => pn_elseif_clauses
end select
select case (char (parse_node_get_rule_key (pn_else_clause)))
case ("else_clause")
    pn_cmd_else => parse_node_get_sub_ptr (pn_else_clause)
    pn_body => parse_node_get_sub_ptr (pn_cmd_else, 2)
    if (associated (pn_body)) then
        allocate (cond%else_body)
        call command_list_compile (cond%else_body, pn_body, global)
    end if
end select
end subroutine cmd_if_compile

```

(Recursively) execute the condition. Context remains global in all cases.

*(CCC Commands: procedures)*+≡

```

recursive subroutine cmd_if_execute (cond, global)
    type(cmd_if_t), intent(inout), target :: cond
    type(rt_data_t), intent(inout), target :: global
    logical :: lval, is_known
    integer :: i
    lval = eval_log (cond%pn_if_lexpr, global%var_list, is_known=is_known)
    if (is_known) then
        if (lval) then

```

```

        if (associated (cond%if_body)) then
            call command_list_execute (cond%if_body, global)
        end if
        return
    end if
else
    call error_undecided ()
    return
end if
if (associated (cond%elsif_cond)) then
    SCAN_ELSEIF: do i = 1, size (cond%elsif_cond)
        lval = eval_log (cond%elsif_cond(i)%pn_if_lexpr, global%var_list, &
            is_known=is_known)
        if (is_known) then
            if (lval) then
                if (associated (cond%elsif_cond(i)%if_body)) then
                    call command_list_execute &
                        (cond%elsif_cond(i)%if_body, global)
                end if
                return
            end if
        else
            call error_undecided ()
            return
        end if
    end do SCAN_ELSEIF
end if
if (associated (cond%else_body)) then
    call command_list_execute (cond%else_body, global)
end if
contains
    subroutine error_undecided ()
        call msg_error ("Undefined result of conditional expression: " &
            // "neither branch will be executed")
    end subroutine error_undecided
end subroutine cmd_if_execute

```

### Include another command-list file

The include command allocates a local parse tree. This must not be deleted before the command object itself is deleted, since pointers may point to subobjects of it.

```

<CCC Commands: types>+≡
    type :: cmd_include_t
    private
        type(string_t) :: file
        type(command_list_t), pointer :: command_list => null ()
        type(parse_tree_t) :: parse_tree
    end type cmd_include_t

```

Finalizer: delete the command list.

```

<CCC Commands: procedures>+≡

```

```

subroutine cmd_include_final (include)
  type(cmd_include_t), intent(inout) :: include
  call parse_tree_final (include%parse_tree)
  if (associated (include%command_list)) then
    call command_list_final (include%command_list)
    deallocate (include%command_list)
  end if
end subroutine cmd_include_final

```

Compile file contents: First parse the file, then immediately compile its contents.  
Use the global data set.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_include_compile (include, pn, global)
  type(cmd_include_t), pointer :: include
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_file
  type(string_t) :: file
  logical :: exist
  integer :: u
  type(stream_t), target :: stream
  type(lexer_t) :: lexer
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  pn_file => parse_node_get_sub_ptr (pn_arg)
  allocate (include)
  file = parse_node_get_string (pn_file)
  inquire (file=char(file), exist=exist)
  if (exist) then
    include%file = file
  else
    include%file = global%os_data%whizard_cutspath // "/" // file
    inquire (file=char(include%file), exist=exist)
    if (.not. exist) then
      call msg_error ("Include file '" // char (file) // "' not found")
      return
    end if
  end if
  u = free_unit ()
  call lexer_init_cmd_list (lexer, global%lexer)
  call stream_init (stream, char (include%file))
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (include%parse_tree, syntax_cmd_list, lexer)
  call stream_final (stream)
  call lexer_final (lexer)
  close (u)
  allocate (include%command_list)
  call command_list_compile &
    (include%command_list, parse_tree_get_root_ptr (include%parse_tree), &
    global)
end subroutine cmd_include_compile

```

Execute file contents in the global context.

*(CCC Commands: procedures)+≡*

```

subroutine cmd_include_execute (include, global)
  type(cmd_include_t), intent(inout), target :: include
  type(rt_data_t), intent(inout), target :: global
  if (associated (include%command_list)) then
    call msg_message &
      ("Including script file '" // char (include%file) // "'")
    call command_list_execute (include%command_list, global)
  end if
end subroutine cmd_include_execute

```

## Quit command execution

The code is the return code of the whole program if it is terminated by this command.

```

<CCC Commands: types>+≡
  type :: cmd_quit_t
  private
  logical :: has_code = .false.
  type(parse_node_t), pointer :: pn_code_expr => null ()
end type cmd_quit_t

```

Finalizer.

```

<CCC Commands: procedures>+≡
  subroutine cmd_quit_final (quit)
    type(cmd_quit_t), intent(inout) :: quit
  end subroutine cmd_quit_final

```

Compile: allocate a quit object which serves as a placeholder.

```

<CCC Commands: procedures>+≡
  subroutine cmd_quit_compile (quit, pn, global)
    type(cmd_quit_t), pointer :: quit
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg
    allocate (quit)
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    if (associated (pn_arg)) then
      quit%pn_code_expr => parse_node_get_sub_ptr (pn_arg)
      quit%has_code = .true.
    end if
  end subroutine cmd_quit_compile

```

Execute: The quit command does not execute anything, it just stops command execution. This is achieved by setting quit flag and quit code in the global variable list. However, the return code, if present, is an expression which has to be evaluated.

```

<CCC Commands: procedures>+≡
  subroutine cmd_quit_execute (quit, global)
    type(cmd_quit_t), intent(inout), target :: quit
    type(rt_data_t), intent(inout), target :: global

```

```

logical :: is_known
if (quit%has_code) then
  global%quit_code = eval_int (quit%pn_code_expr, global%var_list, &
    is_known=is_known)
  if (.not. is_known) then
    call msg_error ("Undefined return code of quit/exit command")
  end if
end if
global%quit = .true.
end subroutine cmd_quit_execute

```

### 20.2.3 The command list

The command list holds a list of commands and relevant global data.

```

<Commands: public>≡
  public :: command_list_t

<Commands: types>+≡
  type :: command_list_t
    ! not private anymore as required by the whizard-c-interface
    class(command_t), pointer :: first => null ()
    class(command_t), pointer :: last => null ()
    contains
    <Commands: command list: TBP>
  end type command_list_t

```

Output.

```

<Commands: command list: TBP>≡
  procedure :: write => command_list_write

<Commands: procedures>+≡
  subroutine command_list_write (cmd_list, unit, indent)
    class(command_list_t), intent(in) :: cmd_list
    integer, intent(in), optional :: unit, indent
    class(command_t), pointer :: cmd
    cmd => cmd_list%first
    do while (associated (cmd))
      call cmd%write (unit, indent)
      cmd => cmd%next
    end do
  end subroutine command_list_write

```

Append a new command to the list and free the original pointer.

```

<Commands: command list: TBP>+≡
  procedure :: append => command_list_append

<Commands: procedures>+≡
  subroutine command_list_append (cmd_list, command)
    class(command_list_t), intent(inout) :: cmd_list
    class(command_t), intent(inout), pointer :: command
    if (associated (cmd_list%last)) then
      cmd_list%last%next => command
    else

```

```

        cmd_list%first => command
    end if
    cmd_list%last => command
    command => null ()
end subroutine command_list_append

```

Finalize.

```

<Commands: command list: TBP>+≡
    procedure :: final => command_list_final

<Commands: procedures>+≡
    recursive subroutine command_list_final (cmd_list)
        class(command_list_t), intent(inout) :: cmd_list
        class(command_t), pointer :: command
        do while (associated (cmd_list%first))
            command => cmd_list%first
            cmd_list%first => cmd_list%first%next
            call command%final ()
            deallocate (command)
        end do
        cmd_list%last => null ()
    end subroutine command_list_final

```

## 20.2.4 Compiling the parse tree

Transform a parse tree into a command list. Initialization is assumed to be done.

After each command, we set a breakpoint.

```

<Commands: command list: TBP>+≡
    procedure :: compile => command_list_compile

<Commands: procedures>+≡
    recursive subroutine command_list_compile (cmd_list, pn, global)
        class(command_list_t), intent(inout), target :: cmd_list
        type(parse_node_t), intent(in), target :: pn
        type(rt_data_t), intent(inout), target :: global
        type(parse_node_t), pointer :: pn_cmd
        class(command_t), pointer :: command
        integer :: i
        pn_cmd => parse_node_get_sub_ptr (pn)
        do i = 1, parse_node_get_n_sub (pn)
            call dispatch_command (command, pn_cmd)
            call command%compile (global)
            call cmd_list%append (command)
            call terminate_now_if_signal ()
            pn_cmd => parse_node_get_next_ptr (pn_cmd)
        end do
    end subroutine command_list_compile

```



## 20.2.5 Executing the command list

Also here, after each command we set a breakpoint.

```
<Commands: command list: TBP>+≡
    procedure :: execute => command_list_execute
<Commands: procedures>+≡
    recursive subroutine command_list_execute (cmd_list, global)
        class(command_list_t), intent(in) :: cmd_list
        type(rt_data_t), intent(inout), target :: global
        class(command_t), pointer :: command
        command => cmd_list%first
        COMMAND_COND: do while (associated (command))
            call command%execute (global)
            call terminate_now_if_signal ()
            if (global%quit) exit COMMAND_COND
            command => command%next
        end do COMMAND_COND
    end subroutine command_list_execute
```

## 20.2.6 Command list syntax

```
<Commands: public>+≡
    public :: syntax_cmd_list
<Commands: variables>≡
    type(syntax_t), target, save :: syntax_cmd_list

<Commands: public>+≡
    public :: syntax_cmd_list_init
<Commands: procedures>+≡
    subroutine syntax_cmd_list_init ()
        type(ifile_t) :: ifile
        call define_cmd_list_syntax (ifile)
        call syntax_init (syntax_cmd_list, ifile)
        call ifile_final (ifile)
    end subroutine syntax_cmd_list_init

<Commands: public>+≡
    public :: syntax_cmd_list_final
<Commands: procedures>+≡
    subroutine syntax_cmd_list_final ()
        call syntax_final (syntax_cmd_list)
    end subroutine syntax_cmd_list_final

<Commands: public>+≡
    public :: syntax_cmd_list_write
<Commands: procedures>+≡
    subroutine syntax_cmd_list_write (unit)
        integer, intent(in), optional :: unit
        call syntax_write (syntax_cmd_list, unit)
    end subroutine syntax_cmd_list_write
```

*<Commands: procedures>+≡*

```

subroutine define_cmd_list_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ command_list = command*")
  call ifile_append (ifile, "ALT command = " &
    // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
    // "cmd_seed | " &
    // "cmd_var | cmd_slha | " &
    // "cmd_show | " &
    // "cmd_expect | " &
    // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
    // "cmd_weight | cmd_selection | cmd_reweight | " &
    // "cmd_beams | cmd_integrate | cmd_me_test | " &
    // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
    // "cmd_clear | cmd_record | " &
    // "cmd_analysis | " &
    // "cmd_unstable | cmd_stable | cmd_simulate | cmd_rescan | " &
    // "cmd_process | cmd_compile | cmd_load | cmd_exec | " &
    // "cmd_scan | cmd_if | cmd_include | cmd_quit | " &
    // "cmd_polarized | cmd_unpolarized | " &
    // "cmd_beam_polarization | " &
    // "cmd_open_out | cmd_close_out | cmd_printd | cmd_printf | " &
    // "cmd_write_analysis | cmd_compile_analysis | " &
    // "cmd_histogram_writer | cmd_plot_writer | cmd_nlo_setup | " &
    // "cmd_set_alpha_qed | cmd_process_sum ")
  call ifile_append (ifile, "GRO options = '{' local_command_list '}'")
  call ifile_append (ifile, "SEQ local_command_list = local_command*")
  call ifile_append (ifile, "ALT local_command = " &
    // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
    // "cmd_seed | " &
    // "cmd_var | cmd_slha | " &
    // "cmd_show | " &
    // "cmd_expect | " &
    // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
    // "cmd_weight | cmd_selection | cmd_reweight | " &
    // "cmd_beams | " &
    // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
    // "cmd_clear | cmd_record | " &
    // "cmd_analysis | " &
    // "cmd_beam_polarization | " &
    // "cmd_open_out | cmd_close_out | cmd_printd | cmd_printf | " &
    // "cmd_write_analysis | cmd_compile_analysis | " &
    // "cmd_histogram_writer | cmd_plot_writer | cmd_nlo_setup | " &
    // "cmd_set_alpha_qed")
  call ifile_append (ifile, "LIS list_of_expr = expr+")
  call ifile_append (ifile, "SEQ expr_pair = expr ':' expr")
  call ifile_append (ifile, "LIS list_of_expr_pairs = expr_pair+")
  call ifile_append (ifile, "KEY soft_mass_regulator")
  call ifile_append (ifile, "SEQ cmd_soft_mass_regulator = " &
    // "soft_mass_regulator '=' expr")
  call ifile_append (ifile, "KEY clear_collinear_mass_regulators")
  call ifile_append (ifile, "SEQ cmd_clear_collinear_mass_regulators = " &
    // "clear_collinear_mass_regulators");
  call ifile_append (ifile, "KEY collinear_mass_regulators")

```

```

call ifile_append (ifile, "SEQ cmd_collinear_mass_regulators = " &
// "collinear_mass_regulators '=' list_of_expr")
call ifile_append (ifile, "KEY charges")
call ifile_append (ifile, "SEQ cmd_charges = charges '=' list_of_expr")
call ifile_append (ifile, "KEY mask")
call ifile_append (ifile, "SEQ cmd_clear_mask = clear_mask")
call ifile_append (ifile, "KEY clear_mask")
call ifile_append (ifile, "SEQ cmd_mask = mask '=' list_of_expr_pairs")
call ifile_append (ifile, "KEY clear_charges")
call ifile_append (ifile, "SEQ cmd_clear_charges = clear_charges")
call ifile_append (ifile, "ALT nlo_setup_command = " &
// "cmd_soft_mass_regulator | cmd_collinear_mass_regulators | " &
// "cmd_charges | cmd_mask | cmd_clear_mask | " &
// "cmd_clear_collinear_mass_regulators | cmd_clear_charges")
call ifile_append (ifile, "KEY alpha_qed")
call ifile_append (ifile, "SEQ cmd_set_alpha_qed = alpha_qed '=' expr")
call ifile_append (ifile, "ALT nlo_setup_local_command = local_command | " &
// "nlo_setup_command")
call ifile_append (ifile, "SEQ nlo_setup_local_command_list = " &
// "nlo_setup_local_command*")
call ifile_append (ifile, "GRO nlo_setup_options = '{' " &
// "nlo_setup_local_command_list '}'")
call ifile_append (ifile, "SEQ cmd_model = model '=' model_name")
call ifile_append (ifile, "KEY model")
call ifile_append (ifile, "ALT model_name = model_id | string_literal")
call ifile_append (ifile, "IDE model_id")
call ifile_append (ifile, "SEQ cmd_library = library '=' lib_name")
call ifile_append (ifile, "KEY library")
call ifile_append (ifile, "ALT lib_name = lib_id | string_literal")
call ifile_append (ifile, "IDE lib_id")
call ifile_append (ifile, "ALT cmd_var = " &
// "cmd_log_decl | cmd_log | " &
// "cmd_int | cmd_real | cmd_complex | cmd_num | " &
// "cmd_string_decl | cmd_string | cmd_alias | " &
// "cmd_result")
call ifile_append (ifile, "SEQ cmd_log_decl = logical cmd_log")
call ifile_append (ifile, "SEQ cmd_log = '?' var_name '=' lexpr")
call ifile_append (ifile, "SEQ cmd_int = int var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_real = real var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_complex = complex var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_num = var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_string_decl = string cmd_string")
call ifile_append (ifile, "SEQ cmd_string = " &
// "'$' var_name '=' sexpr") ! $
call ifile_append (ifile, "SEQ cmd_alias = alias var_name '=' cexpr")
call ifile_append (ifile, "SEQ cmd_result = result '=' expr")
call ifile_append (ifile, "SEQ cmd_slha = slha_action slha_arg options?")
call ifile_append (ifile, "ALT slha_action = " &
// "read_slha | write_slha")
call ifile_append (ifile, "KEY read_slha")
call ifile_append (ifile, "KEY write_slha")
call ifile_append (ifile, "ARG slha_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_show = show show_arg?")
call ifile_append (ifile, "KEY show")

```

```

call ifile_append (ifile, "ARG show_arg = ( var_generic* )")
call ifile_append (ifile, "ALT var_generic = " &
// "model | beams | results | unstable | real | int | " &
// "cuts | weight | scale | factorization_scale | renormalization_scale | " &
// "analysis | expect | " &
// "library_spec | " &
// "intrinsic | result_var | " &
// "log_var | alias_var | string_var | num_var | alpha_qed")
call ifile_append (ifile, "KEY results")
call ifile_append (ifile, "KEY intrinsic")
call ifile_append (ifile, "SEQ library_spec = library lib_name?")
call ifile_append (ifile, "SEQ result_var = result_key result_arg?")
call ifile_append (ifile, "SEQ log_var = '?' log_var_spec?")
call ifile_append (ifile, "ALT log_var_spec = log_val | variable")
call ifile_append (ifile, "GRO log_val = ( lexpr )")
call ifile_append (ifile, "SEQ alias_var = alias alias_var_spec?")
call ifile_append (ifile, "ALT alias_var_spec = alias_val | variable")
call ifile_append (ifile, "GRO alias_val = ( cexpr )")
call ifile_append (ifile, "SEQ string_var = '$' string_var_spec?") ! $
call ifile_append (ifile, "ALT string_var_spec = string_val | variable")
call ifile_append (ifile, "GRO string_val = ( sexpr )")
call ifile_append (ifile, "SEQ num_var = num_var_spec")
call ifile_append (ifile, "ALT num_var_spec = num_val | variable")
call ifile_append (ifile, "GRO num_val = ( expr )")
call ifile_append (ifile, "SEQ cmd_expect = expect expect_arg options?")
call ifile_append (ifile, "KEY expect")
call ifile_append (ifile, "ARG expect_arg = ( lexpr )")
call ifile_append (ifile, "SEQ cmd_cuts = cuts '=' lexpr")
call ifile_append (ifile, "SEQ cmd_scale = scale '=' expr")
call ifile_append (ifile, "SEQ cmd_fac_scale = factorization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_ren_scale = renormalization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_weight = weight '=' expr")
call ifile_append (ifile, "SEQ cmd_selection = selection '=' lexpr")
call ifile_append (ifile, "SEQ cmd_reweight = reweight '=' expr")
call ifile_append (ifile, "KEY cuts")
call ifile_append (ifile, "KEY scale")
call ifile_append (ifile, "KEY factorization_scale")
call ifile_append (ifile, "KEY renormalization_scale")
call ifile_append (ifile, "KEY weight")
call ifile_append (ifile, "KEY selection")
call ifile_append (ifile, "KEY reweight")
call ifile_append (ifile, "SEQ cmd_process = process process_id '=' " &
// "process_prt '=' process_prt nlo_setup_options?")
call ifile_append (ifile, "KEY process")
call ifile_append (ifile, "KEY '='>")
call ifile_append (ifile, "LIS process_prt = cexpr+")
call ifile_append (ifile, "KEY process_sum")
call ifile_append (ifile, "SEQ cmd_process_sum = process_sum process_id " &
// "'=' process_id '+' process_id nlo_setup_options?")
call ifile_append (ifile, "KEY nlo_setup")
call ifile_append (ifile, "SEQ cmd_nlo_setup = nlo_setup " &
// "process_id nlo_setup_options?")
call ifile_append (ifile, "SEQ cmd_compile = compile_cmd options?")
call ifile_append (ifile, "SEQ compile_cmd = compile_clause compile_arg?")

```

```

call ifile_append (ifile, "SEQ compile_clause = compile exec_name_spec?")
call ifile_append (ifile, "KEY compile")
call ifile_append (ifile, "SEQ exec_name_spec = as exec_name")
call ifile_append (ifile, "KEY as")
call ifile_append (ifile, "ALT exec_name = exec_id | string_literal")
call ifile_append (ifile, "IDE exec_id")
call ifile_append (ifile, "ARG compile_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_load = load load_arg?")
call ifile_append (ifile, "KEY load")
call ifile_append (ifile, "ARG load_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_exec = exec exec_arg")
call ifile_append (ifile, "KEY exec")
call ifile_append (ifile, "ARG exec_arg = ( sexpr )")
call ifile_append (ifile, "SEQ cmd_beams = beams '=' beam_def")
call ifile_append (ifile, "KEY beams")
call ifile_append (ifile, "SEQ beam_def = beam_spec strfun_seq*")
call ifile_append (ifile, "SEQ beam_spec = beam_list options?")
call ifile_append (ifile, "LIS beam_list = cexpr, cexpr?")
call ifile_append (ifile, "SEQ strfun_seq = '>' strfun_pair")
call ifile_append (ifile, "LIS strfun_pair = strfun_def, strfun_def?")
call ifile_append (ifile, "SEQ strfun_def = strfun_id options?")
call ifile_append (ifile, "ALT strfun_id = " &
    // "none | lhpdf | isr | epa | ewa | pdf_builtin | " &
    // "circe1 | circe2 | energy_scan | beam_events | " &
    // "user_sf_spec")
call ifile_append (ifile, "KEY none")
call ifile_append (ifile, "KEY lhpdf")
call ifile_append (ifile, "KEY isr")
call ifile_append (ifile, "KEY epa")
call ifile_append (ifile, "KEY ewa")
call ifile_append (ifile, "KEY circe1")
call ifile_append (ifile, "KEY circe2")
call ifile_append (ifile, "KEY energy_scan")
call ifile_append (ifile, "KEY beam_events")
call ifile_append (ifile, "KEY pdf_builtin")
call ifile_append (ifile, "SEQ user_sf_spec = user_strfun user_arg")
call ifile_append (ifile, "KEY user_strfun")
call ifile_append (ifile, "SEQ cmd_integrate = " &
    // "integrate proc_arg options?")
call ifile_append (ifile, "KEY integrate")
call ifile_append (ifile, "ARG proc_arg = ( proc_id* )")
call ifile_append (ifile, "IDE proc_id")
call ifile_append (ifile, "SEQ cmd_me_test = " &
    // "matrix_element_test proc_arg1 options?")
call ifile_append (ifile, "KEY matrix_element_test")
call ifile_append (ifile, "ARG proc_arg1 = ( proc_id )")
call ifile_append (ifile, "SEQ cmd_seed = seed '=' expr")
call ifile_append (ifile, "KEY seed")
call ifile_append (ifile, "SEQ cmd_iterations = " &
    // "iterations '=' iterations_list")
call ifile_append (ifile, "KEY iterations")
call ifile_append (ifile, "LIS iterations_list = iterations_spec+")
call ifile_append (ifile, "ALT iterations_spec = it_spec")
call ifile_append (ifile, "SEQ it_spec = expr calls_spec adapt_spec?")

```

```

call ifile_append (ifile, "SEQ calls_spec = ':' expr")
call ifile_append (ifile, "SEQ adapt_spec = ':' sexpr")
call ifile_append (ifile, "SEQ cmd_sample_format = " &
// "sample_format '=' event_format_list")
call ifile_append (ifile, "KEY sample_format")
call ifile_append (ifile, "LIS event_format_list = event_format+")
call ifile_append (ifile, "IDE event_format")
call ifile_append (ifile, "SEQ cmd_observable = " &
// "observable analysis_tag options?")
call ifile_append (ifile, "KEY observable")
call ifile_append (ifile, "SEQ cmd_histogram = " &
// "histogram analysis_tag histogram_arg " &
// "options?")
call ifile_append (ifile, "KEY histogram")
call ifile_append (ifile, "ARG histogram_arg = (expr, expr, expr?)")
call ifile_append (ifile, "SEQ cmd_plot = plot analysis_tag options?")
call ifile_append (ifile, "KEY plot")
call ifile_append (ifile, "SEQ cmd_graph = graph graph_term '=' graph_def")
call ifile_append (ifile, "KEY graph")
call ifile_append (ifile, "SEQ graph_term = analysis_tag options?")
call ifile_append (ifile, "SEQ graph_def = graph_term graph_append*")
call ifile_append (ifile, "SEQ graph_append = '&' graph_term")
call ifile_append (ifile, "SEQ cmd_analysis = analysis '=' lexpr")
call ifile_append (ifile, "KEY analysis")
call ifile_append (ifile, "SEQ cmd_open_out = open_out open_arg options?")
call ifile_append (ifile, "SEQ cmd_close_out = close_out open_arg options?")
call ifile_append (ifile, "KEY open_out")
call ifile_append (ifile, "KEY close_out")
call ifile_append (ifile, "ARG open_arg = (sexpr)")
call ifile_append (ifile, "SEQ cmd_printd = printd_cmd options?")
call ifile_append (ifile, "SEQ printd_cmd = printd sprintf_args?")
call ifile_append (ifile, "KEY printd")
call ifile_append (ifile, "SEQ cmd_printf = printf_cmd options?")
call ifile_append (ifile, "SEQ printf_cmd = printf_clause sprintf_args?")
call ifile_append (ifile, "SEQ printf_clause = printf sexpr")
call ifile_append (ifile, "KEY printf")
call ifile_append (ifile, "SEQ cmd_clear = clear clear_arg?")
call ifile_append (ifile, "KEY clear")
call ifile_append (ifile, "ARG clear_arg = ( clear_obj* )")
call ifile_append (ifile, "ALT clear_obj = " &
// "iterations | cuts | weight | scale | factorization_scale | " &
// "renormalization_scale | analysis | expect | " &
// "analysis_tag")
call ifile_append (ifile, "SEQ cmd_record = record_cmd")
call ifile_append (ifile, "SEQ cmd_unstable = " &
// "unstable unstable_list options?")
call ifile_append (ifile, "KEY unstable")
call ifile_append (ifile, "LIS unstable_list = unstable_decl+")
call ifile_append (ifile, "SEQ unstable_decl = cexpr unstable_arg")
call ifile_append (ifile, "ARG unstable_arg = ( proc_id+ )")
call ifile_append (ifile, "SEQ cmd_stable = stable stable_list options?")
call ifile_append (ifile, "KEY stable")
call ifile_append (ifile, "LIS stable_list = cexpr+")
call ifile_append (ifile, "KEY polarized")

```

```

call ifile_append (ifile, "SEQ cmd_polarized = polarized polarized_list options?")
call ifile_append (ifile, "LIS polarized_list = cexpr+")
call ifile_append (ifile, "KEY unpolarized")
call ifile_append (ifile, "SEQ cmd_unpolarized = unpolarized unpolarized_list options?")
call ifile_append (ifile, "LIS unpolarized_list = cexpr+")
call ifile_append (ifile, "SEQ cmd_simulate = " &
    // "simulate proc_arg options?")
call ifile_append (ifile, "KEY simulate")
call ifile_append (ifile, "SEQ cmd_rescan = " &
    // "rescan sexpr proc_arg options?")
call ifile_append (ifile, "KEY rescan")
call ifile_append (ifile, "SEQ cmd_scan = scan_spec scan_body?")
call ifile_append (ifile, "SEQ scan_spec = scan scan_command?")
call ifile_append (ifile, "KEY scan")
call ifile_append (ifile, "ALT scan_command = " &
    // "cmd_model_list | cmd_library_list | " &
    // "cmd_seed_list | " &
    // "cmd_cuts_list | cmd_scale_list | " &
    // "cmd_fac_scale_list | cmd_ren_scale_list | " &
    // "cmd_weight_list | cmd_selection_list | " &
    // "cmd_reweight_list | cmd_analysis_list | " &
    // "cmd_var_list")
call ifile_append (ifile, "SEQ cmd_model_list = model = model_list_arg")
call ifile_append (ifile, "ARG model_list_arg = ( model_name* )")
call ifile_append (ifile, "SEQ cmd_library_list = " &
    // "library = library_list_arg")
call ifile_append (ifile, "ARG library_list_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_seed_list = seed = num_step_list_arg")
call ifile_append (ifile, "ARG num_step_list_arg = ( num_steps_expr* )")
call ifile_append (ifile, "ALT num_steps_expr = grouped_num_steps | num_steps")
call ifile_append (ifile, "GRO grouped_num_steps = ( num_steps )")
call ifile_append (ifile, "SEQ num_steps = expr range_spec?")
call ifile_append (ifile, "SEQ range_spec = '='>' expr step_spec?")
call ifile_append (ifile, "SEQ step_spec = step_op expr")
call ifile_append (ifile, "ALT step_op = " &
    // "'/' | '/'-' | '/*' | '//' | '+/+' | '*/'*")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '/'-")
call ifile_append (ifile, "KEY '/*'")
call ifile_append (ifile, "KEY '//")
call ifile_append (ifile, "KEY '+/+")
call ifile_append (ifile, "KEY '*/'")
call ifile_append (ifile, "ALT cmd_var_list = " &
    // "log_decl_list | log_list | " &
    // "int_list | real_list | complex_list | num_list | " &
    // "string_decl_list | string_list | alias_list")
call ifile_append (ifile, "SEQ log_decl_list = logical log_list")
call ifile_append (ifile, "SEQ log_list = '?' var_name '=' log_list_arg")
call ifile_append (ifile, "ARG log_list_arg = ( lexpr* )")
call ifile_append (ifile, "SEQ int_list = " &
    // "int var_name '=' num_step_list_arg")
call ifile_append (ifile, "SEQ real_list = " &
    // "real var_name '=' num_step_list_arg")
call ifile_append (ifile, "SEQ complex_list = " &

```

```

// "complex var_name '=' num_step_list_arg")
call ifile_append (ifile, "SEQ num_list = var_name '=' num_step_list_arg")
call ifile_append (ifile, "SEQ string_decl_list = string string_list")
call ifile_append (ifile, "SEQ string_list = " &
// "'$' var_name '=' string_list_arg") !$
call ifile_append (ifile, "ARG string_list_arg = ( sexpr* )")
call ifile_append (ifile, "SEQ alias_list = " &
// "alias var_name alias_list_arg")
call ifile_append (ifile, "ARG alias_list_arg = ( cexpr* )")
call ifile_append (ifile, "SEQ cmd_cuts_list = cuts = log_list_arg")
call ifile_append (ifile, "SEQ cmd_scale_list = scale = num_list_arg")
call ifile_append (ifile, "SEQ cmd_fac_scale_list = factorization_scale = num_list_arg")
call ifile_append (ifile, "SEQ cmd_ren_scale_list = renormalization_scale = num_list_arg")
call ifile_append (ifile, "SEQ cmd_weight_list = weight = num_list_arg")
call ifile_append (ifile, "SEQ cmd_selection_list = selection = log_list_arg")
call ifile_append (ifile, "SEQ cmd_reweight_list = reweight = num_list_arg")
call ifile_append (ifile, "ARG num_list_arg = ( expr* )")
call ifile_append (ifile, "SEQ cmd_analysis_list = analysis = log_list_arg")
call ifile_append (ifile, "GRO scan_body = '{' command_list '}'")
call ifile_append (ifile, "SEQ cmd_if = " &
// "if lexpr then command_list elsif_clauses else_clause endif")
call ifile_append (ifile, "SEQ elsif_clauses = cmd_elseif*")
call ifile_append (ifile, "SEQ cmd_elseif = elsif lexpr then command_list")
call ifile_append (ifile, "SEQ else_clause = cmd_else?")
call ifile_append (ifile, "SEQ cmd_else = else command_list")
call ifile_append (ifile, "SEQ cmd_include = include include_arg")
call ifile_append (ifile, "KEY include")
call ifile_append (ifile, "ARG include_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_quit = quit_cmd quit_arg?")
call ifile_append (ifile, "ALT quit_cmd = quit | exit")
call ifile_append (ifile, "KEY quit")
call ifile_append (ifile, "KEY exit")
call ifile_append (ifile, "ARG quit_arg = ( expr )")
call ifile_append (ifile, "SEQ cmd_beam_polarization = " &
// "beam_polarization '=' bp_mode options?")
call ifile_append (ifile, "KEY beam_polarization")
call ifile_append (ifile, "ALT bp_mode = off | bp_defs")
call ifile_append (ifile, "KEY off")
call ifile_append (ifile, "LIS bp_defs = bp_def, bp_def?")
call ifile_append (ifile, "ARG arg_unary = ( expr )")
call ifile_append (ifile, "ARG arg_binary = (expr, expr)")
call ifile_append (ifile, "ARG arg_tenary = (expr, expr, expr)")
call ifile_append (ifile, "ALT bp_def = " &
// "none | bp_circ | bp_trans | bp_axis | bp_long | bp_diag | " &
// "bp_dens")
call ifile_append (ifile, "SEQ bp_circ = circular arg_unary")
call ifile_append (ifile, "KEY circular")
call ifile_append (ifile, "SEQ bp_trans = transverse arg_binary")
call ifile_append (ifile, "KEY transverse")
call ifile_append (ifile, "SEQ bp_axis = axis arg_tenary")
call ifile_append (ifile, "KEY axis")
call ifile_append (ifile, "SEQ bp_long = longitudinal arg_unary")
call ifile_append (ifile, "KEY longitudinal")
call ifile_append (ifile, "SEQ bp_dens = density_matrix arg_binary")

```



```

call ifile_append (ifile, "KEY density_matrix")
call ifile_append (ifile, "SEQ bp_diag = diagonal_density bp_diag_args")
call ifile_append (ifile, "KEY diagonal_density")
call ifile_append (ifile, "ARG bp_diag_args = ( bp_diag_entry+ )")
call ifile_append (ifile, "SEQ bp_diag_entry = expr ':' expr")
call ifile_append (ifile, "SEQ cmd_write_analysis = " &
// "write_analysis_clause options?")
call ifile_append (ifile, "SEQ cmd_compile_analysis = " &
// "compile_analysis_clause options?")
call ifile_append (ifile, "SEQ write_analysis_clause = " &
// "write_analysis write_analysis_arg?")
call ifile_append (ifile, "SEQ compile_analysis_clause = " &
// "compile_analysis write_analysis_arg?")
call ifile_append (ifile, "KEY write_analysis")
call ifile_append (ifile, "KEY compile_analysis")
call ifile_append (ifile, "ARG write_analysis_arg = ( analysis_tag* )")
call ifile_append (ifile, "SEQ cmd_histogram_writer = " &
// "histogram_writer '=' writer_macro")
call ifile_append (ifile, "KEY histogram_writer")
call ifile_append (ifile, "SEQ cmd_plot_writer = " &
// "plot_writer '=' writer_macro")
call ifile_append (ifile, "KEY plot_writer")
call ifile_append (ifile, "GRO writer_macro = '{' command_list '}'")
call define_expr_syntax (ifile, particles=.true., analysis=.true.)
end subroutine define_cmd_list_syntax

```

*<Commands: public>+≡*

```
public :: lexer_init_cmd_list
```

*<Commands: procedures>+≡*

```

subroutine lexer_init_cmd_list (lexer, parent_lexer)
type(lexer_t), intent(out) :: lexer
type(lexer_t), intent(in), optional, target :: parent_lexer
call lexer_init (lexer, &
comment_chars = "#!", &
quote_chars = "'", &
quote_match = "'", &
single_chars = "()[]{};,:%?$@", &
special_class = (/ "+-*/^", "<>=~ " /) , &
keyword_list = syntax_get_keyword_list_ptr (syntax_cmd_list), &
parent = parent_lexer)
end subroutine lexer_init_cmd_list

```

## 20.2.7 Unit Tests

*<Commands: public>+≡*

```
public :: commands_test
```

*<Commands: tests>≡*

```

subroutine commands_test (u, results)
integer, intent(in) :: u
type(test_results_t), intent(inout) :: results
<Commands: execute tests>

```

```
end subroutine commands_test
```

## Prepare Sindarin code

This routine parses an internal file, prints the parse tree, and returns a parse node to the root. We use the routine in the tests below.

```
<Commands: tests>+≡
subroutine parse_ifile (ifile, pn_root, u)
  type(ifile_t), intent(in) :: ifile
  type(parse_node_t), pointer, intent(out) :: pn_root
  integer, intent(in), optional :: u
  type(stream_t), target :: stream
  type(lexer_t), target :: lexer
  type(parse_tree_t) :: parse_tree

  call lexer_init_cmd_list (lexer)
  call stream_init (stream, ifile)
  call lexer_assign_stream (lexer, stream)

  call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
  if (present (u)) call parse_tree_write (parse_tree, u)
  pn_root => parse_tree_get_root_ptr (parse_tree)

  call stream_final (stream)
  call lexer_final (lexer)
end subroutine parse_ifile
```

## Empty command list

Compile and execute an empty command list. Should do nothing but test the integrity of the workflow.

```
<Commands: execute tests>≡
  call test (commands_1, "commands_1", &
    "empty command list", &
    u, results)

<Commands: tests>+≡
subroutine commands_1 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root

  write (u, "(A)")  "* Test output: commands_1"
  write (u, "(A)")  "* Purpose: compile and execute empty command list"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")
```

```

call syntax_cmd_list_init ()
call global%global_init ()

write (u, "(A)")  "* Parse empty file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"

if (associated (pn_root)) then
  call command_list%compile (pn_root, global)
end if

write (u, "(A)")
write (u, "(A)")  "* Execute command list"

call command_list%execute (global)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_1"

end subroutine commands_1

```

## Read model

Execute a model assignment.

```

<Commands: execute tests>+≡
  call test (commands_2, "commands_2", &
    "model", &
    u, results)

<Commands: tests>+≡
subroutine commands_2 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root

  write (u, "(A)")  "* Test output: commands_2"
  write (u, "(A)")  "* Purpose: set model"
  write (u, "(A)")

```

```

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')

call ifile_write (ifile, u)

write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)
write (u, "(A)")  "global model = '" &
    // char (global%model%get_name ()) // "' "

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_2"

end subroutine commands_2

```

## Declare Process

Read a model, then declare a process. The process library is allocated explicitly. For the process definition, We take the default (**omega**) method. Since we do not compile, O'MEGA is not actually called.

```
(Commands: execute tests)+≡
  call test (commands_3, "commands_3", &
    "process declaration", &
    u, results)

(Commands: tests)+≡
  subroutine commands_3 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_3"
    write (u, "(A)")  "* Purpose: define process"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()
    call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
      .false., is_known = .true.)

    allocate (lib)
    call lib%init (var_str ("lib_cmd3"))
    call global%add_prclib (lib)

    write (u, "(A)")  "* Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = "Test"')
    call ifile_append (ifile, 'process t3 = s, s => s, s')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
```

```

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%prclib_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_3"

end subroutine commands_3

```

## Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

```

<Commands: execute tests>+≡
  call test (commands_4, "commands_4", &
    "compilation", &
    u, results)

<Commands: tests>+≡
  subroutine commands_4 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_4"
    write (u, "(A)")  "* Purpose: define process and compile library"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()

```

```

call syntax_model_file_init ()
call global%global_init ()
call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd4"))
call global%add_prclib (lib)

write (u, "(A)")  "*  Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process t4 = s, s => s, s')
call ifile_append (ifile, 'compile ("lib_cmd4")')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "*  Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "*  Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%prclib_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "*  Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "*  Test output end: commands_4"

end subroutine commands_4

```

## Integrate Process

Read a model, then declare a process, compile the library, and integrate over phase space. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

```
(Commands: execute tests)+≡
    call test (commands_5, "commands_5", &
               "integration", &
               u, results)

(Commands: tests)+≡
    subroutine commands_5 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root
        type(prclib_entry_t), pointer :: lib

        write (u, "(A)")  "* Test output: commands_5"
        write (u, "(A)")  "*   Purpose: define process, iterations, and integrate"
        write (u, "(A)")

        write (u, "(A)")  "*   Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
        call syntax_model_file_init ()
        call global%global_init ()
        call var_list_set_string (global%var_list, var_str ("method"), &
                                  var_str ("unit_test"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("phs_method"), &
                                  var_str ("single"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("integration_method"), &
                                  var_str ("midpoint"), is_known=.true.)
        call var_list_set_real (global%var_list, var_str ("sqrtts"), &
                                 1000._default, is_known=.true.)

        allocate (lib)
        call lib%init (var_str ("lib_cmd5"))
        call global%add_prclib (lib)

        write (u, "(A)")  "*   Input file"
        write (u, "(A)")

        call ifile_append (ifile, 'model = "Test"')
        call ifile_append (ifile, 'process t5 = s, s => s, s')
        call ifile_append (ifile, 'compile')
        call ifile_append (ifile, 'iterations = 1:1000')
        call ifile_append (ifile, 'integrate (t5)')

        call ifile_write (ifile, u)

        write (u, "(A)")
        write (u, "(A)")  "*   Parse file"
```



```

write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call reset_interaction_counter ()
call command_list%execute (global)

call global%it_list%write (u)
write (u, "(A)")
call global%process_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_5"

end subroutine commands_5

```

## Variables

Set intrinsic and user-defined variables.

```

<Commands: execute tests>+≡
  call test (commands_6, "commands_6", &
    "variables", &
    u, results)

<Commands: tests>+≡
  subroutine commands_6 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

```

```

write (u, "(A)")  "* Test output: commands_6"
write (u, "(A)")  "* Purpose: define and set variables"
write (u, "(A)")

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()
call global%write_vars (u, [ &
    var_str ("$_run_id"), &
    var_str ("?unweighted"), &
    var_str ("sqrts")])

write (u, "(A)")
write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, '$_run_id = "run1"')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'sqrts = 1000')
call ifile_append (ifile, 'int j = 10')
call ifile_append (ifile, 'real x = 1000.')
call ifile_append (ifile, 'complex z = 5')
call ifile_append (ifile, 'string $text = "abcd"')
call ifile_append (ifile, 'logical ?flag = true')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write_vars (u, [ &
    var_str ("$_run_id"), &
    var_str ("?unweighted"), &
    var_str ("sqrts"), &
    var_str ("j"), &
    var_str ("x"), &

```

```

        var_str ("z"), &
        var_str ("text"), &
        var_str ("flag"]])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_6"

end subroutine commands_6

```

## Process library

Open process libraries explicitly.

```

(Commands: execute tests) +=
    call test (commands_7, "commands_7", &
        "process library", &
        u, results)

(Commands: tests) +=
    subroutine commands_7 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root

        write (u, "(A)")  "* Test output: commands_7"
        write (u, "(A)")  "* Purpose: declare process libraries"
        write (u, "(A)")

        write (u, "(A)")  "* Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
        call global%global_init ()
        call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
            .false., is_known = .true.)
        global%os_data%fc = "Fortran-compiler"
        global%os_data%fcflags = "Fortran-flags"

        write (u, "(A)")
        write (u, "(A)")  "* Input file"
        write (u, "(A)")

```

```

call ifile_append (ifile, 'library = "lib_cmd7_1"')
call ifile_append (ifile, 'library = "lib_cmd7_2"')
call ifile_append (ifile, 'library = "lib_cmd7_1"')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write_libraries (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_7"

end subroutine commands_7

```

## Generate events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

```

<Commands: execute tests>+≡
  call test (commands_8, "commands_8", &
    "event generation", &
    u, results)

<Commands: tests>+≡
  subroutine commands_8 (u)

```

```

integer, intent(in) :: u
type(ifile_t) :: ifile
type(command_list_t), target :: command_list
type(rt_data_t), target :: global
type(parse_node_t), pointer :: pn_root
type(prclib_entry_t), pointer :: lib

write (u, "(A)")  "* Test output: commands_8"
write (u, "(A)")  "* Purpose: define process, integrate, generate events"
write (u, "(A)")

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call var_list_set_string (global%var_list, var_str ("method"), &
    var_str ("unit_test"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("phs_method"), &
    var_str ("single"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("integration_method"), &
    var_str ("midpoint"), is_known=.true.)
call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd8"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_8_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_8_p)')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'n_events = 3')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'simulate (commands_8_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

```

```

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
!   write (u, "(A)")

call command_list%execute (global)

!   call global%process_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_8"

end subroutine commands_8

```

## Define cuts

Declare a cut expression.

```

<Commands: execute tests>+≡
  call test (commands_9, "commands_9", &
    "cuts", &
    u, results)

<Commands: tests>+≡
  subroutine commands_9 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(string_t), dimension(0) :: no_vars

    write (u, "(A)")  "* Test output: commands_9"
    write (u, "(A)")  "* Purpose: define cuts"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()

```

```

write (u, "(A)")  "*  Input file"
write (u, "(A)")

call ifile_append (ifile, 'cuts = all Pt > 0 [particle]')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "*  Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "*  Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write (u, vars = no_vars)

write (u, "(A)")
write (u, "(A)")  "*  Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "*  Test output end: commands_9"

end subroutine commands_9

```

## Beams

Define beam setup.

```

<Commands: execute tests>+≡
  call test (commands_10, "commands_10", &
    "beams", &
    u, results)

<Commands: tests>+≡
  subroutine commands_10 (u)

```

```

integer, intent(in) :: u
type(ifile_t) :: ifile
type(command_list_t), target :: command_list
type(rt_data_t), target :: global
type(parse_node_t), pointer :: pn_root
type(string_t), dimension(0) :: no_vars

write (u, "(A)")  "* Test output: commands_10"
write (u, "(A)")  "* Purpose: define beams"
write (u, "(A)")

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = QCD')
call ifile_append (ifile, 'sqrts = 1000')
call ifile_append (ifile, 'beams = p, p')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write_beams (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()

```



```

call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_10"

end subroutine commands_10

```

## Structure functions

Define beam setup with structure functions

```

(Commands: execute tests) +=
  call test (commands_11, "commands_11", &
    "structure functions", &
    u, results)

(Commands: tests) +=
  subroutine commands_11 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(string_t), dimension(0) :: no_vars

    write (u, "(A)")  "* Test output: commands_11"
    write (u, "(A)")  "* Purpose: define beams with structure functions"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()

    write (u, "(A)")  "* Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = QCD')
    call ifile_append (ifile, 'sqrts = 1100')
    call ifile_append (ifile, 'beams = p, p => lhpdf => pdf_builtin, isr')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")

```

```

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write_beams (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_11"

end subroutine commands_11

```

## Rescan events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration. Then, rescan the generated event sample.

```

<Commands: execute tests>+≡
  call test (commands_12, "commands_12", &
    "event rescanning", &
    u, results)

<Commands: tests>+≡
  subroutine commands_12 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_12"
    write (u, "(A)")  "* Purpose: generate events and rescan"
    write (u, "(A)")

```

```

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call var_list_set_string (global%var_list, var_str ("method"), &
    var_str ("unit_test"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("phs_method"), &
    var_str ("single"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("integration_method"), &
    var_str ("midpoint"), is_known=.true.)
call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd12"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_12_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_12_p)')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'n_events = 3')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'simulate (commands_12_p)')
call ifile_append (ifile, '?write_raw = false')
call ifile_append (ifile, 'rescan "commands_12_p" (commands_12_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
!   write (u, "(A)")

call command_list%execute (global)

```

```

!      call global%process_stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_12"

end subroutine commands_12

```

## Event Files

Set output formats for event files.

```

<Commands: execute tests>+≡
      call test (commands_13, "commands_13", &
        "event output formats", &
        u, results)

<Commands: tests>+≡
      subroutine commands_13 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root
        type(prclib_entry_t), pointer :: lib
        logical :: exist

        write (u, "(A)")  "* Test output: commands_13"
        write (u, "(A)")  "*   Purpose: generate events and rescan"
        write (u, "(A)")

        write (u, "(A)")  "*   Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
        call syntax_model_file_init ()
        call global%global_init ()
        call var_list_set_string (global%var_list, var_str ("method"), &
          var_str ("unit_test"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("phs_method"), &
          var_str ("single"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("integration_method"), &
          var_str ("midpoint"), is_known=.true.)
        call var_list_set_real (global%var_list, var_str ("sqrts"), &

```

```

1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd13"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_13_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_13_p)')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'n_events = 1')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'sample_format = weight_stream')
call ifile_append (ifile, 'simulate (commands_13_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
!   write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")
write (u, "(A)")  "* Verify output files"
write (u, "(A)")

inquire (file = "commands_13_p.evx", exist = exist)
if (exist) write (u, "(1x,A)")  "raw"

inquire (file = "commands_13_p.weights.dat", exist = exist)
if (exist) write (u, "(1x,A)")  "weight_stream"

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

```

```

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_13"

end subroutine commands_13

```

## Compile Empty Libraries

(This is a regression test:) Declare two empty libraries and compile them.

```

<Commands: execute tests>+≡
call test (commands_14, "commands_14", &
  "empty libraries", &
  u, results)

<Commands: tests>+≡
subroutine commands_14 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  type(prclib_entry_t), pointer :: lib

  write (u, "(A)")  "* Test output: commands_14"
  write (u, "(A)")  "* Purpose: define and compile empty libraries"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "* Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'library = "lib1"')
  call ifile_append (ifile, 'library = "lib2"')
  call ifile_append (ifile, 'compile ()')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

```

```

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%prclib_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_14"

end subroutine commands_14

```

## Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

```

<Commands: execute tests>+≡
  call test (commands_15, "commands_15", &
    "compilation", &
    u, results)

<Commands: tests>+≡
  subroutine commands_15 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_15"
    write (u, "(A)")  "* Purpose: define process and compile library"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

```

```

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call var_list_set_string (global%var_list, var_str ("$method"), &
    var_str ("unit_test"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
    var_str ("single"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"), &
    var_str ("midpoint"), is_known=.true.)
call var_list_set_real (global%var_list, var_str ("sqrts"), &
    1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd15"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process t15 = s, s => s, s')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (t15)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%prclib_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
call global%final ()

```



```

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_15"

end subroutine commands_15

```

## 20.3 Toplevel module WHIZARD

```

⟨whizard.f90⟩≡
⟨File header⟩

module whizard

  ⟨Use file utils⟩
  ⟨Use strings⟩
  use limits, only: VERSION_STRING !NODEP!
  use limits, only: EOF, BACKSLASH !NODEP!
  use diagnostics !NODEP!
  use unit_tests
  use ifiles
  use formats
  use md5
  use os_interface
  use lexers
  use parser
  use colors
  use state_matrices
  use analysis
  use variables
  use user_code_interface
  use expressions
  use particles
  use models
  use sorting
  use evaluators
  use phs_forests
  ! use beams
  ! use polarizations
  ! use processes
  ! use decays
  use sm_qcd
  use sf_aux
  use sf_mappings
  use sf_base
  use sf_pdf_builtin
  use sf_lhapdf
  use sf_circe1
  use sf_circe2
  use sf_lumi_linker
  use sf_isr
  use sf_epa
  use sf_ewa

```

```

use sf_escan
use sf_beam_events
use sf_user
use phs_base
use phs_single
use phs_wood
use rng_base
use rng_tao
use mci_base
use mci_midpoint
use mci_vamp
use prclib_interfaces
use particle_specifiers
use process_libraries
use prclib_stacks
! use hepmc_interface
! use interactions
! use strfun
! use slha_interface
! use cascades
! use events
! use blha_driver
! use blha_config
! use vamp !NODEP!
use prc_test
use prc_omega
use subvt_expr
use processes
use process_stacks
use events

use eio_data
use eio_base
use eio_raw
use eio_lhef
use eio_hepmc
use eio_weights

use iterations
use beam_structures
use rt_data
use dispatch
use process_configurations
use compilations
use integrations
use event_streams
use simulations

use commands

<Standard module head>

<WHIZARD: public>

```

```

⟨WHIZARD: types⟩

⟨WHIZARD: variables⟩

    save

contains

⟨WHIZARD: procedures⟩

end module whizard

```

### 20.3.1 Options

Here we introduce a wrapper that holds various user options, so they can transparently be passed from the main program to the `whizard` object. Most parameters are used for initializing the `global` state.

```

⟨WHIZARD: public⟩≡
    public :: whizard_options_t
⟨WHIZARD: types⟩≡
    type :: whizard_options_t
        type(string_t) :: preload_model
        type(string_t) :: preload_library
    logical :: rebuild_library = .false.
    ! logical :: rebuild_user
    logical :: rebuild_phs = .false.
    logical :: rebuild_grids = .false.
    logical :: rebuild_events = .false.
end type whizard_options_t

```

### 20.3.2 The whizard object

An object of type `whizard_t` is the top-level wrapper for a `WHIZARD` instance. The object holds various default settings and the current state of the generator, the `global` object of type `rt_data_t`. This object contains, for instance, the list of variables and the process libraries.

Since components of the `global` subobject are frequently used as targets, the `whizard` object should also consistently carry the `target` attribute.

The various self-tests do not use this object. They initialize only specific subsets of the system, according to their needs.

Note: we intend to allow several concurrent instances. In the current implementation, there are still a few obstacles to this: the model library and the syntax tables are global variables, and the error handling uses global state. This should be improved.

```

⟨WHIZARD: public⟩+≡
    public :: whizard_t

```

```

<WHIZARD: types>+=
  type :: whizard_t
    type(whizard_options_t) :: options
    type(rt_data_t) :: global
  contains
    <WHIZARD: whizard: TBP>
  end type whizard_t

```

### 20.3.3 Initialization and finalization

```

<WHIZARD: whizard: TBP>=
  procedure :: init => whizard_init

<WHIZARD: procedures>=
  subroutine whizard_init (whizard, options)
    class(whizard_t), intent(out), target :: whizard
    type(whizard_options_t), intent(in) :: options
    call init_syntax_tables ()
    whizard%options = options
    call whizard%global%global_init ()
    call whizard%init_rebuild_flags ()
    call whizard%preload_model ()
    call whizard%preload_library ()
  end subroutine whizard_init

<WHIZARD: whizard: TBP>+=
  procedure :: final => whizard_final

<WHIZARD: procedures>+=
  subroutine whizard_final (whizard)
    class(whizard_t), intent(inout), target :: whizard
    call whizard%global%final ()
    call final_syntax_tables ()
  end subroutine whizard_final

```

Set the rebuild flags. They can be specified on the command line and set the initial value for the associated logical variables.

```

<WHIZARD: whizard: TBP>+=
  procedure :: init_rebuild_flags => whizard_init_rebuild_flags

<WHIZARD: procedures>+=
  subroutine whizard_init_rebuild_flags (whizard)
    class(whizard_t), intent(inout), target :: whizard
    associate (var_list => whizard%global%var_list, options => whizard%options)
      call var_list_append_log &
        (var_list, var_str ("?rebuild_library"), options%rebuild_library, &
          intrinsic=.true.)
      call var_list_append_log &
        (var_list, var_str ("?rebuild_phase_space"), options%rebuild_phs, &
          intrinsic=.true.)
      call var_list_append_log &
        (var_list, var_str ("?rebuild_grids"), options%rebuild_grids, &
          intrinsic=.true.)
    end associate
  end subroutine whizard_init_rebuild_flags

```

```

        call var_list_append_log &
            (var_list, var_str ("?rebuild_events"), options%rebuild_events, &
            intrinsic=.true.)
    end associate
end subroutine whizard_init_rebuild_flags

```

This procedure preloads a model, if a model name is given.

```

<WHIZARD: whizard: TBP>+≡
    procedure :: preload_model => whizard_preload_model

<WHIZARD: procedures>+≡
    subroutine whizard_preload_model (whizard)
        class(whizard_t), intent(inout), target :: whizard
        type(string_t) :: model_name, filename
        type(model_t), pointer :: model
        type(var_list_t), pointer :: model_vars
        model_name = whizard%options%preload_model
        if (model_name /= "") then
            associate (var_list => whizard%global%var_list)
                filename = model_name // ".mdl"
                call model_list_read_model &
                    (model_name, filename, whizard%global%os_data, model)
                if (associated (model)) then
                    model_vars => model_get_var_list_ptr (model)
                    call var_list_init_copies (var_list, model_vars)
                    call var_list_synchronize &
                        (var_list, model_vars, reset_pointers = .true.)
                    call msg_message ("Using model: " &
                        // char (model_get_name (model)))
                    call var_list_set_string (var_list, var_str ("$model_name"), &
                        model_get_name (model), is_known=.true.)
                    whizard%global%model => model
                else
                    call msg_fatal ("Preloading model " // char (model_name) &
                        // " failed")
                end if
            end associate
        else
            call msg_message ("No model preloaded")
        end if
    end subroutine whizard_preload_model

```

This procedure preloads a library, if a library name is given.

Note: This version just opens a new library with that name. It does not load (yet) an existing library on file, as previous WHIZARD versions would do.

```

<WHIZARD: whizard: TBP>+≡
    procedure :: preload_library => whizard_preload_library

<WHIZARD: procedures>+≡
    subroutine whizard_preload_library (whizard)
        class(whizard_t), intent(inout), target :: whizard
        type(string_t) :: library_name, filename
        type(prclib_entry_t), pointer :: lib_entry
        type(process_library_t), pointer :: lib

```

```

library_name = whizard%options%preload_library
if (library_name /= "") then
  allocate (lib_entry)
  call lib_entry%init (library_name)
  call whizard%global%add_prclib (lib_entry)
  call msg_message ("Preloaded library: " // char (library_name))
else
  call msg_message ("No library preloaded")
end if
end subroutine whizard_preload_library

```

### 20.3.4 Initialization and finalization (old version)

These procedures initialize and finalize global variables. Most of them are collected in the `global` data record located here, the others are syntax tables located in various modules, which do not change during program execution. Furthermore, there is a global model list and a global process store, which get filled during program execution but are finalized here.

During initialization, we can preload a default model and initialize a default library for setting up processes. The default library is loaded if requested by the setup. Further libraries can be loaded as specified by command-line flags.

```

<XXX WHIZARD: public>≡
  public :: global

<XXX WHIZARD: variables>≡
  !!! JRR: WK please check
  type(rt_data_t), target :: global

<XXX WHIZARD: public>+≡
  public :: whizard_init

<XXX WHIZARD: procedures>≡
  subroutine whizard_init &
    (preload_model, preload_libs, default_lib, &
     rebuild_library, rebuild_user, &
     rebuild_phs, rebuild_grids, rebuild_events, &
     recompile_library, &
     time_estimate, &
     paths, &
     user_code_enable, &
     n_user_src, user_src, &
     n_user_lib, user_lib, &
     user_target)
    type(string_t), intent(in) :: preload_model, preload_libs
    type(string_t), intent(in) :: default_lib
    logical, intent(in) :: rebuild_library, rebuild_user
    logical, intent(in) :: rebuild_phs, rebuild_grids
    logical, intent(in) :: rebuild_events
    logical, intent(in) :: recompile_library
    logical, intent(in) :: time_estimate
    type(paths_t), intent(in), optional :: paths
    integer, intent(in), optional :: n_user_src, n_user_lib
    logical, intent(in), optional :: user_code_enable

```

```

type(string_t), intent(in), optional :: user_src, user_lib, user_target
type(string_t) :: filename, libname, libs
type(var_list_t), pointer :: model_vars
logical :: user
integer :: n_src, n_lib
type(string_t), dimension(:), allocatable :: src, lib
!!! JRR: WK please check
type(prclib_entry_t), pointer :: lib_entry
integer :: i
!!! JRR: WK please check
call global%global_init (paths)
user = .false.
if (present (user_code_enable)) user = user_code_enable
n_src = 0
if (present (n_user_src) .and. present (user_src)) n_src = n_user_src
n_lib = 0
if (present (n_user_lib) .and. present (user_lib)) n_lib = n_user_lib
if (user) then
    call splice (user_src, n_src, src)
    call splice (user_lib, n_lib, lib)
    call user_code_init (src, lib, user_target, rebuild_user, global%os_data)
end if
! call var_list_append_log &
!     (global%var_list, var_str ("?rebuild_library"), rebuild_library, &
!         intrinsic=.true.)
! call var_list_append_log &
!     (global%var_list, var_str ("?rebuild_phase_space"), rebuild_phs, &
!         intrinsic=.true.)
! call var_list_append_log &
!     (global%var_list, var_str ("?rebuild_grids"), rebuild_grids, &
!         intrinsic=.true.)
! call var_list_append_log &
!     (global%var_list, var_str ("?rebuild_events"), rebuild_events, &
!         intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?recompile_library"), recompile_library, &
        intrinsic=.true.)
! call var_list_append_log &
!     (global%var_list, var_str ("?time_estimate"), time_estimate, &
!         intrinsic=.true.)
call init_syntax_tables ()
!!! JRR: WK please check
!!! Q: What about the preloaded libraries and the libmanager now?
! call process_library_store_load_static &
!     (global%os_data, global%prc_lib, global%model, global%var_list)
libs = adjustl (preload_libs)
SCAN_LIBS: do while (libs /= "")
    call split (libs, libname, " ")
    allocate (lib_entry)
    call lib_entry%init (libname)
    call global%add_prclib (lib_entry)
!     call process_library_store_append &
!         (libname, global%os_data, global%prc_lib)
!     call process_library_load &

```

```

!           (global%prc_lib, global%os_data, global%model, global%var_list, &
!           ignore=.true.)
end do SCAN_LIBS
if (.not. associated (global%prclib_stack%first)) then
    allocate (lib_entry)
    call lib_entry%init (default_lib)
    call global%add_prclib (lib_entry)
end if
!!! Q: What to do with the rebuild_library option?
!   if (.not. associated (global%prc_lib)) then
!       call process_library_store_append &
!       (default_lib, global%os_data, global%prc_lib)
!       if (.not. (rebuild_library .or. recompile_library)) then
!           call process_library_load (global%prc_lib, &
!           global%os_data, global%model, global%var_list, ignore=.true.)
!       else
!           call var_list_set_string (global%var_list, &
!           var_str ("library_name"), &
!           process_library_get_name (global%prc_lib), is_known=.true.)
!       end if
!   end if
!!! JRR: WK please check
filename = preload_model // ".mdl"
call model_list_read_model &
    (preload_model, filename, global%os_data, global%model)
if (associated (global%model)) then
    model_vars => model_get_var_list_ptr (global%model)
    call var_list_init_copies (global%var_list, model_vars)
    call var_list_synchronize &
    (global%var_list, model_vars, reset_pointers = .true.)
    call msg_message ("Using model: " &
    // char (model_get_name (global%model)))
    call var_list_set_string (global%var_list, var_str ("model_name"), &
    model_get_name (global%model), is_known=.true.)
end if
contains
subroutine splice (string, n, array)
    type(string_t), intent(in) :: string
    integer :: n
    type(string_t), dimension(:), allocatable :: array
    integer :: i
    type(string_t) :: buffer
    allocate (array (n))
    buffer = string
    do i = 1, n
        call split (buffer, array(i), " ")
    end do
end subroutine splice
end subroutine whizard_init

```

Initialize/finalize the syntax tables used by WHIZARD:

```

<WHIZARD: public>+≡
public :: init_syntax_tables

```



```

    public :: final_syntax_tables
<WHIZARD: procedures>+≡
    subroutine init_syntax_tables ()
        call syntax_model_file_init ()
        call syntax_phs_forest_init ()
        call syntax_pexpr_init ()
!       call syntax_slha_init ()
        call syntax_cmd_list_init ()
!       call syntax_blha_contract_init ()
    end subroutine init_syntax_tables

    subroutine final_syntax_tables ()
        call syntax_model_file_final ()
        call syntax_phs_forest_final ()
        call syntax_pexpr_final ()
!       call syntax_slha_final ()
        call syntax_cmd_list_final ()
!       call syntax_blha_contract_final ()
    end subroutine final_syntax_tables

```

Write the syntax tables to external files.

```

<XXX WHIZARD: public>+≡
    public :: write_syntax_tables
<XXX WHIZARD: procedures>+≡
    subroutine write_syntax_tables ()
        integer :: unit
        character(*), parameter :: file_model = "whizard.model_file.syntax"
        character(*), parameter :: file_phs = "whizard.phase_space_file.syntax"
        character(*), parameter :: file_pexpr = "whizard.prt_expressions.syntax"
        character(*), parameter :: file_slha = "whizard.slha.syntax"
        character(*), parameter :: file_sindarin = "whizard.sindarin.syntax"
        unit = free_unit ()
        print *, "Writing file '" // file_model // "'"
        open (unit=unit, file=file_model, status="replace", action="write")
        write (unit, "(A)")  VERSION_STRING
        write (unit, "(A)")  "Syntax definition file: " // file_model
        call syntax_model_file_write (unit)
        close (unit)
        print *, "Writing file '" // file_phs // "'"
        open (unit=unit, file=file_phs, status="replace", action="write")
        write (unit, "(A)")  VERSION_STRING
        write (unit, "(A)")  "Syntax definition file: " // file_phs
        call syntax_phs_forest_write (unit)
        close (unit)
        print *, "Writing file '" // file_pexpr // "'"
        open (unit=unit, file=file_pexpr, status="replace", action="write")
        write (unit, "(A)")  VERSION_STRING
        write (unit, "(A)")  "Syntax definition file: " // file_pexpr
        call syntax_pexpr_write (unit)
        close (unit)
        print *, "Writing file '" // file_slha // "'"
        open (unit=unit, file=file_slha, status="replace", action="write")
        write (unit, "(A)")  VERSION_STRING

```

```

write (unit, "(A)") "Syntax definition file: " // file_slha
call syntax_slha_write (unit)
close (unit)
print *, "Writing file '" // file_sindarin // "'"
open (unit=unit, file=file_sindarin, status="replace", action="write")
write (unit, "(A)") VERSION_STRING
write (unit, "(A)") "Syntax definition file: " // file_sindarin
call syntax_cmd_list_write (unit)
close (unit)
end subroutine write_syntax_tables

```

Apart from the global data which have been initialized above, the process and model lists need to be finalized.

```

<XXX WHIZARD: public>+≡
public :: whizard_final

<XXX WHIZARD: procedures>+≡
subroutine whizard_final ()
!   call rt_data_global_final (global)
!   call user_code_final ()
!   call decay_store_final ()
!   call process_store_final ()
!   call model_list_final ()
!   call final_syntax_tables ()
end subroutine whizard_final

```

### 20.3.5 Execute command lists

Process commands given on the command line, stored as an `ifile`. The whole input is read, compiled and executed as a whole.

```

<XXX WHIZARD: public>+≡
public :: whizard_process_ifile

<XXX WHIZARD: procedures>+≡
subroutine whizard_process_ifile (ifile, quit, quit_code)
type(ifile_t), intent(in) :: ifile
logical, intent(out) :: quit
integer, intent(out) :: quit_code
type(lexer_t), target :: lexer
type(stream_t), target :: stream
call msg_message ("Reading commands given on the command line")
call lexer_init_cmd_list (lexer)
call stream_init (stream, ifile)
call whizard_process_stream (stream, lexer, quit, quit_code)
call stream_final (stream)
call lexer_final (lexer)
end subroutine whizard_process_ifile

```

Process standard input as a command list. The whole input is read, compiled and executed as a whole.

```

<XXX WHIZARD: public>+≡
public :: whizard_process_stdin

```

```

<XXX WHIZARD: procedures>+≡
  subroutine whizard_process_stdin (quit, quit_code)
    logical, intent(out) :: quit
    integer, intent(out) :: quit_code
    type(lexer_t), target :: lexer
    type(stream_t), target :: stream
    call msg_message ("Reading commands from standard input")
    call lexer_init_cmd_list (lexer)
    call stream_init (stream, 5)
    call whizard_process_stream (stream, lexer, quit, quit_code)
    call stream_final (stream)
    call lexer_final (lexer)
  end subroutine whizard_process_stdin

```

Process a file as a command list.

```

<WHIZARD: whizard: TBP>+≡
  procedure :: process_file => whizard_process_file

<WHIZARD: procedures>+≡
  subroutine whizard_process_file (whizard, file, quit, quit_code)
    class(whizard_t), intent(inout), target :: whizard
    type(string_t), intent(in) :: file
    logical, intent(out) :: quit
    integer, intent(out) :: quit_code
    integer :: u
    type(lexer_t), target :: lexer
    type(stream_t), target :: stream
    logical :: exist
    call msg_message ("Reading commands from file '" // char (file) // "'")
    inquire (file=char(file), exist=exist)
    if (exist) then
      u = free_unit ()
      call lexer_init_cmd_list (lexer)
      call stream_init (stream, char (file))
      call whizard%process_stream (stream, lexer, quit, quit_code)
      call stream_final (stream)
      call lexer_final (lexer)
    else
      call msg_error ("File '" // char (file) // "' not found")
    end if
  end subroutine whizard_process_file

```

```

<WHIZARD: whizard: TBP>+≡
  procedure :: process_stream => whizard_process_stream

<WHIZARD: procedures>+≡
  subroutine whizard_process_stream (whizard, stream, lexer, quit, quit_code)
    class(whizard_t), intent(inout), target :: whizard
    type(stream_t), intent(inout), target :: stream
    type(lexer_t), intent(inout), target :: lexer
    logical, intent(out) :: quit
    integer, intent(out) :: quit_code
    type(parse_tree_t) :: parse_tree
    type(command_list_t), target :: command_list

```

```

call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
if (associated (parse_tree_get_root_ptr (parse_tree))) then
    call command_list%compile (parse_tree_get_root_ptr (parse_tree), &
        whizard%global)
end if
call command_list%execute (whizard%global)
call command_list%final ()
quit = whizard%global%quit
quit_code = whizard%global%quit_code
end subroutine whizard_process_stream

```

### 20.3.6 The WHIZARD shell

This procedure implements interactive mode. One line is processed at a time.

```

<XXX WHIZARD: public>+≡
    public :: whizard_shell

<XXX WHIZARD: procedures>+≡
    subroutine whizard_shell (quit_code)
        integer, intent(out) :: quit_code
        type(lexer_t), target :: lexer
        type(stream_t), target :: stream
        type(string_t) :: prompt1
        type(string_t) :: prompt2
        type(string_t) :: input
        type(string_t) :: extra
        integer :: last
        integer :: iostat
        logical :: mask_tmp
        logical :: quit
        call msg_message ("Launching interactive shell")
        call lexer_init_cmd_list (lexer)
        prompt1 = "whish? "
        prompt2 = "      > "
        COMMAND_LOOP: do
            call put (6, prompt1)
            call get (5, input, iostat=iostat)
            if (iostat > 0 .or. iostat == EOF) exit COMMAND_LOOP
            CONTINUE_INPUT: do
                last = len_trim (input)
                if (extract (input, last, last) /= BACKSLASH) exit CONTINUE_INPUT
                call put (6, prompt2)
                call get (5, extra, iostat=iostat)
                if (iostat > 0) exit COMMAND_LOOP
                input = replace (input, last, extra)
            end do CONTINUE_INPUT
            call stream_init (stream, input)
            mask_tmp = mask_fatal_errors
            mask_fatal_errors = .true.
            call whizard_process_stream (stream, lexer, quit, quit_code)
            msg_count = 0
            mask_fatal_errors = mask_tmp

```

```

        call stream_final (stream)
        if (quit) exit COMMAND_LOOP
    end do COMMAND_LOOP
    print *
    call lexer_final (lexer)
end subroutine whizard_shell

```

### 20.3.7 Self-tests

This is for developers only, but needs a well-defined interface.

```

<WHIZARD: public>+≡
    public :: whizard_check

<WHIZARD: procedures>+≡
    subroutine whizard_check (check, lhpdf_present, results)
        type(string_t), intent(in) :: check
        logical, intent(in) :: lhpdf_present
        type(lexer_t), target :: lexer
        type(test_results_t), intent(inout) :: results
        integer :: u
        u = free_unit ()
        open (u, file="whizard_check." // char (check) // ".log", &
            action="write", status="replace")
!       global%lexer => lexer
        call msg_message (repeat ('=', 76), 0)
        call msg_message ("Running self-test: " // char (check), 0)
        call msg_message (repeat ('-', 76), 0)
        select case (char (check))
!       case ("analysis"); call analysis_test ()
!       case ("beams"); call beam_test ()
!       case ("cascades"); call cascade_test ()
!       case ("colors"); call color_test ()
!       case ("commands"); call command_test ()
!       case ("evaluators"); call evaluator_test (global%model)
!       case ("events"); call event_test ()
!       case ("expressions"); call expressions_test ()
!       case ("formats"); call format_test ()
!       case ("hepmc"); call hepmc_test ()
!       case ("interactions"); call interaction_test ()
!       case ("lexers"); call lexer_test (lexer, 6)
!       case ("md5"); call md5_test ()
!       case ("models"); call models_test ()
!       case ("os_interface"); call os_interface_test ()
!       case ("parser"); call parse_test ()
!       case ("particles"); call particles_test ()
!       case ("phs_forests"); call phs_forest_test ()
!       case ("polarizations"); call polarization_test ()
        case ("sm_qcd")
            call sm_qcd_test (u, results)
        case ("sf_aux")
            call sf_aux_test (u, results)
        case ("sf_mappings")
            call sf_mappings_test (u, results)
    end subroutine whizard_check

```

```

case ("sf_base")
    call sf_base_test (u, results)
case ("sf_pdf_builtin")
    call sf_pdf_builtin_test (u, results)
case ("sf_lhapdf")
    if (lhpdf_present) then
        call sf_lhapdf_test (u, results)
    end if
case ("sf_isr")
    call sf_isr_test (u, results)
case ("sf_epa")
    call sf_epa_test (u, results)
case ("sf_ewa")
    call sf_ewa_test (u, results)
case ("phs_base")
    call phs_base_test (u, results)
case ("phs_single")
    call phs_single_test (u, results)
case ("phs_wood")
    call phs_wood_test (u, results)
case ("mci_base")
    call mci_base_test (u, results)
case ("rng_base")
    call rng_base_test (u, results)
case ("rng_tao")
    call rng_tao_test (u, results)
case ("mci_midpoint")
    call mci_midpoint_test (u, results)
case ("mci_vamp")
    call mci_vamp_test (u, results)
case ("prclib_interfaces")
    call prclib_interfaces_test (u, results)
case ("particle_specifiers")
    call particle_specifiers_test (u, results)
case ("process_libraries")
    call process_libraries_test (u, results)
case ("prclib_stacks")
    call prclib_stacks_test (u, results)
!   case ("slha_interface"); call slha_test ()
!   case ("sorting"); call sorting_test ()
!   case ("state_matrices"); call state_matrix_test ()
!! !! case ("strfun"); call strfun_test (lhpdf_present)
!   case ("blha"); call blha_test (global%model)
case ("prc_test")
    call prc_test_test (u, results)
case ("subevt_expr")
    call subevt_expr_test (u, results)
case ("processes")
    call processes_test (u, results)
case ("process_stacks")
    call process_stacks_test (u, results)
case ("events")
    call events_test (u, results)
case ("prc_omega")

```

```

        call prc_omega_test (u, results)
case ("eio_data")
    call eio_data_test (u, results)
case ("eio_base")
    call eio_base_test (u, results)
case ("eio_raw")
    call eio_raw_test (u, results)
case ("eio_lhef")
    call eio_lhef_test (u, results)
case ("eio_hepmc")
    call eio_hepmc_test (u, results)
case ("eio_weights")
    call eio_weights_test (u, results)
case ("iterations")
    call iterations_test (u, results)
case ("beam_structures")
    call beam_structures_test (u, results)
case ("rt_data")
    call rt_data_test (u, results)
case ("dispatch")
    call dispatch_test (u, results)
case ("process_configurations")
    call process_configurations_test (u, results)
case ("compilations")
    call compilations_test (u, results)
case ("integrations")
    call integrations_test (u, results)
case ("event_streams")
    call event_streams_test (u, results)
case ("simulations")
    call simulations_test (u, results)
case ("commands")
    call commands_test (u, results)
case ("all")
    call sm_qcd_test (u, results)
    call sf_aux_test (u, results)
    call sf_mappings_test (u, results)
    call sf_base_test (u, results)
    call sf_pdf_builtin_test (u, results)
    if (lhpdf_present) then
        call sf_lhpdf_test (u, results)
    end if
    call phs_base_test (u, results)
    call phs_single_test (u, results)
    call phs_wood_test (u, results)
    call rng_base_test (u, results)
    call rng_tao_test (u, results)
    call mci_base_test (u, results)
    call mci_midpoint_test (u, results)
    call mci_vamp_test (u, results)
    call prclib_interfaces_test (u, results)
    call particle_specifiers_test (u, results)
    call process_libraries_test (u, results)
    call prclib_stacks_test (u, results)

```

```

        call prc_test_test (u, results)
        call subevt_expr_test (u, results)
        call processes_test (u, results)
        call process_stacks_test (u, results)
        call events_test (u, results)
        call prc_omega_test (u, results)
        call eio_data_test (u, results)
        call eio_base_test (u, results)
        call eio_raw_test (u, results)
        call eio_lhef_test (u, results)
        call eio_hePMC_test (u, results)
        call eio_weights_test (u, results)
        call iterations_test (u, results)
        call beam_structures_test (u, results)
        call rt_data_test (u, results)
        call dispatch_test (u, results)
        call process_configurations_test (u, results)
        call compilations_test (u, results)
        call integrations_test (u, results)
        call event_streams_test (u, results)
        call simulations_test (u, results)
        call commands_test (u, results)
    case default
        call msg_fatal ("Self-test '" // char (check) // "' not implemented.")
    end select
    close (u)
end subroutine whizard_check

```

## 20.4 Driver program

The main program handles command options, initializes the environment, and runs WHIZARD in a particular mode (interactive, file, standard input).

```

<Limits: public parameters>+≡
    integer, parameter, public :: CMDLINE_ARG_LEN = 1000

<main.f90>≡
    <File header>

    program main

    <Use strings>
        use system_dependencies !NODEP!
        use limits, only: CMDLINE_ARG_LEN !NODEP!
        use diagnostics !NODEP!
        use unit_tests
        use ifiles
        use os_interface
        use whizard

    implicit none

    ! Main program variable declarations

```



```

character(CMDLINE_ARG_LEN) :: arg
character(2) :: option
type(string_t) :: long_option, value
integer :: i, j, arg_len, arg_status
logical :: look_for_options
logical :: interactive
logical :: banner
type(string_t) :: files, this, model, library, logfile
type(string_t) :: check, checks
type(test_results_t) :: test_results
logical :: success
logical :: user_code_enable = .false.
integer :: n_user_src = 0, n_user_lib = 0
type(string_t) :: user_src, user_lib, user_target
type(paths_t) :: paths
logical :: rebuild_library, rebuild_user
logical :: rebuild_phs, rebuild_grids, rebuild_events
logical :: recompile_library
logical :: time_estimate
type(ifile_t) :: commands
type(string_t) :: command

type(whizard_options_t), allocatable :: options
type(whizard_t), allocatable, target :: whizard_instance

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
library = "default_lib"
banner = .true.
logging = .true.
logfile = "whizard.log"
! libraries = ""
check = ""
checks = ""
user_src = ""
user_lib = ""
user_target = ""
rebuild_library = .false.
rebuild_user = .false.
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
recompile_library = .false.
time_estimate = .true.
call paths_init (paths)

! Read and process options

```

```

i = 0
SCAN_CMDLINE: do
  i = i + 1
  call get_command_argument (i, arg, arg_len, arg_status)
  select case (arg_status)
  case (0)
  case (-1)
    call msg_error (" Command argument truncated: '" // arg // "'")
  case default
    exit SCAN_CMDLINE
  end select
  if (look_for_options) then
    select case (arg(1:2))
    case ("--")
      value = trim (arg)
      call split (value, long_option, "=")
      select case (char (long_option))
      case ("--version")
        call no_option_value (long_option, value)
        call print_version (); stop
      case ("--help")
        call no_option_value (long_option, value)
        call print_usage (); stop
      case ("--prefix")
        paths%prefix = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--exec-prefix")
        paths%exec_prefix = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--bindir")
        paths%bindir = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--libdir")
        paths%libdir = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--includedir")
        paths%includedir = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--datarootdir")
        paths%datarootdir = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--libtool")
        paths%libtool = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--lhapdfdir")
        paths%lhapdfdir = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
      case ("--check")
        check = get_option_value (i, long_option, value)
        checks = checks // " " // check
        cycle SCAN_CMDLINE
      case ("--execute")
        command = get_option_value (i, long_option, value)
        call ifile_append (commands, command)
    end select
  end if
end do

```

```

        cycle SCAN_CMDLINE
case ("--interactive")
    call no_option_value (long_option, value)
    interactive = .true.
    cycle SCAN_CMDLINE
case ("--library")
    library = get_option_value (i, long_option, value)
!    libraries = libraries // " " // library
    cycle SCAN_CMDLINE
case ("--no-library")
    call no_option_value (long_option, value)
    library = ""
    cycle SCAN_CMDLINE
case ("--localprefix")
    paths%localprefix = get_option_value (i, long_option, value)
    cycle SCAN_CMDLINE
case ("--logfile")
    logfile = get_option_value (i, long_option, value)
    cycle SCAN_CMDLINE
case ("--no-logfile")
    call no_option_value (long_option, value)
    logfile = ""
    cycle SCAN_CMDLINE
case ("--logging")
    call no_option_value (long_option, value)
    logging = .true.
    cycle SCAN_CMDLINE
case ("--no-logging")
    call no_option_value (long_option, value)
    logging = .false.
    cycle SCAN_CMDLINE
case ("--banner")
    call no_option_value (long_option, value)
    banner = .true.
    cycle SCAN_CMDLINE
case ("--no-banner")
    call no_option_value (long_option, value)
    banner = .false.
    cycle SCAN_CMDLINE
case ("--model")
    model = get_option_value (i, long_option, value)
    cycle SCAN_CMDLINE
case ("--no-model")
    call no_option_value (long_option, value)
    model = ""
    cycle SCAN_CMDLINE
case ("--rebuild")
    call no_option_value (long_option, value)
    rebuild_library = .true.
    rebuild_user = .true.
    rebuild_phs = .true.
    rebuild_grids = .true.
    rebuild_events = .true.
    cycle SCAN_CMDLINE

```

```

case ("--no-rebuild")
    call no_option_value (long_option, value)
    rebuild_library = .false.
    rebuild_user = .false.
    rebuild_phs = .false.
    rebuild_grids = .false.
    rebuild_events = .false.
    cycle SCAN_CMDLINE
case ("--rebuild-library")
    call no_option_value (long_option, value)
    rebuild_library = .true.
    cycle SCAN_CMDLINE
case ("--rebuild-user")
    call no_option_value (long_option, value)
    rebuild_user = .true.
    cycle SCAN_CMDLINE
case ("--rebuild-phase-space")
    call no_option_value (long_option, value)
    rebuild_phs = .true.
    cycle SCAN_CMDLINE
case ("--rebuild-grids")
    call no_option_value (long_option, value)
    rebuild_grids = .true.
    cycle SCAN_CMDLINE
case ("--rebuild-events")
    call no_option_value (long_option, value)
    rebuild_events = .true.
    cycle SCAN_CMDLINE
case ("--recompile")
    call no_option_value (long_option, value)
    recompile_library = .true.
    rebuild_grids = .true.
    cycle SCAN_CMDLINE
case ("--time-estimate")
    call no_option_value (long_option, value)
    time_estimate = .true.
    cycle SCAN_CMDLINE
case ("--no-time-estimate")
    call no_option_value (long_option, value)
    time_estimate = .false.
    cycle SCAN_CMDLINE
case ("--user")
    user_code_enable = .true.
    cycle SCAN_CMDLINE
case ("--user-src")
    if (user_src == "") then
        user_src = get_option_value (i, long_option, value)
    else
        user_src = user_src // " " &
            // get_option_value (i, long_option, value)
    end if
    n_user_src = n_user_src + 1
    cycle SCAN_CMDLINE
case ("--user-lib")

```

```

        if (user_lib == "") then
            user_lib = get_option_value (i, long_option, value)
        else
            user_lib = user_lib // " " &
                // get_option_value (i, long_option, value)
        end if
        n_user_lib = n_user_lib + 1
        cycle SCAN_CMDLINE
    case ("--user-target")
        user_target = get_option_value (i, long_option, value)
        cycle SCAN_CMDLINE
!   case ("--write-syntax-tables")
!       call no_option_value (long_option, value)
!       call init_syntax_tables ()
!       call write_syntax_tables ()
!       call final_syntax_tables ()
!       stop
!       cycle SCAN_CMDLINE
    case default
        call print_usage ()
        call msg_fatal ("Option '" // trim (arg) // "' not recognized")
    end select
end select
select case (arg(1:1))
case ("-")
    j = 1
    if (len_trim (arg) == 1) then
        look_for_options = .false.
    else
        SCAN_SHORT_OPTIONS: do
            j = j + 1
            if (j > len_trim (arg)) exit SCAN_SHORT_OPTIONS
            option = "-" // arg(j:j)
            select case (option)
            case ("-V")
                call print_version (); stop
            case ("-?", "-h")
                call print_usage (); stop
            case ("-e")
                command = get_option_value (i, var_str (option))
                call ifile_append (commands, command)
                cycle SCAN_CMDLINE
            case ("-i")
                interactive = .true.
                cycle SCAN_SHORT_OPTIONS
            case ("-l")
                if (j == len_trim (arg)) then
                    library = get_option_value (i, var_str (option))
                else
                    library = trim (arg(j+1:))
                end if
                cycle SCAN_CMDLINE
            case ("-L")
                if (j == len_trim (arg)) then

```

```

        logfile = get_option_value (i, var_str (option))
    else
        logfile = trim (arg(j+1:))
    end if
    cycle SCAN_CMDLINE
case ("-m")
    if (j < len_trim (arg)) call msg_fatal &
        ("Option '" // option // "' needs a value")
    model = get_option_value (i, var_str (option))
    cycle SCAN_CMDLINE
case ("-r")
    rebuild_library = .true.
    rebuild_user = .true.
    rebuild_phs = .true.
    rebuild_grids = .true.
    rebuild_events = .true.
    cycle SCAN_SHORT_OPTIONS
case ("-u")
    user_code_enable = .true.
    cycle SCAN_SHORT_OPTIONS
case default
    call print_usage ()
    call msg_fatal &
        ("Option '" // option // "' not recognized")
end select
end do SCAN_SHORT_OPTIONS
end if
case default
    files = files // " " // trim (arg)
end select
else
    files = files // " " // trim (arg)
end if
end do SCAN_CMDLINE

! Overall initialization
if (logfile /= "") call logfile_init (logfile)
! call mask_term_signals ()
if (banner) call msg_banner ()

! Run any self-checks (and no commands)
if (checks /= "") then
    checks = trim (adjustl (checks))
    RUN_CHECKS: do while (checks /= "")
        call split (checks, check, " ")
        call whizard_check (check, LHAPDF_AVAILABLE, test_results)
    end do RUN_CHECKS
    call test_results%wrapup (6, success)
    if (.not. success) quit_code = 7
    quit = .true.
end if

! Set options and initialize the whizard object
allocate (options)

```

```

options%preload_model = model
options%preload_library = library
options%rebuild_library = rebuild_library
! options%rebuild_user = rebuild_user
options%rebuild_phs = rebuild_phs
options%rebuild_grids = rebuild_grids
options%rebuild_events = rebuild_events

allocate (whizard_instance)
call whizard_instance%init (options)

! call whizard_init &
!   (preload_model=model, preload_libs=libraries, default_lib=libname, &
!   rebuild_library=rebuild_library, &
!   rebuild_user=rebuild_user, &
!   rebuild_phs=rebuild_phs, &
!   rebuild_grids=rebuild_grids, &
!   rebuild_events=rebuild_events, &
!   recompile_library=recompile_library, &
!   time_estimate=time_estimate, &
!   paths=paths, &
!   user_code_enable=user_code_enable, &
!   n_user_src=n_user_src, user_src=user_src, &
!   n_user_lib=n_user_lib, user_lib=user_lib, &
!   user_target=user_target)

! ! Run commands given on the command line
! if (.not. quit .and. ifile_get_length (commands) > 0) then
!   call whizard_process_ifile (commands, quit, quit_code)
! end if
!
! if (.not. quit) then
!   ! Process commands from standard input
!   if (.not. interactive .and. files == "") then
!     call whizard_process_stdin (quit, quit_code)
!   !
!   ! ... or process commands from file
!   else
!     files = trim (adjustl (files))
!     SCAN_FILES: do while (files /= "")
!       call split (files, this, " ")
!       call whizard_instance%process_file (this, quit, quit_code)
!       if (quit) exit SCAN_FILES
!     end do SCAN_FILES
!   end if
! end if

! ! Enter an interactive shell if requested
! if (.not. quit .and. interactive) then
!   call whizard_shell (quit_code)
! end if
!

```

```

!   ! Overall finalization
!   call ifile_final (commands)

deallocate (options)

call whizard_instance%final ()
deallocate (whizard_instance)

!   call terminate_now_if_signal ()
!   call release_term_signals ()
call msg_terminate (quit_code = quit_code)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
contains

subroutine no_option_value (option, value)
  type(string_t), intent(in) :: option, value
  if (value /= "") then
    call msg_error (" Option '" // char (option) // "' should have no value")
  end if
end subroutine no_option_value

function get_option_value (i, option, value) result (string)
  type(string_t) :: string
  integer, intent(inout) :: i
  type(string_t), intent(in) :: option
  type(string_t), intent(in), optional :: value
  character(CMDLINE_ARG_LEN) :: arg
  character(CMDLINE_ARG_LEN) :: arg_value
  integer :: arg_len, arg_status
  logical :: has_value
  if (present (value)) then
    has_value = value /= ""
  else
    has_value = .false.
  end if
  if (has_value) then
    string = value
  else
    i = i + 1
    call get_command_argument (i, arg_value, arg_len, arg_status)
    select case (arg_status)
    case (0)
    case (-1)
      call msg_error (" Option value truncated: '" // arg // "'")
    case default
      call print_usage ()
      call msg_fatal (" Option '" // char (option) // "' needs a value")
    end select
    select case (arg(1:1))
    case ("-")
      call print_usage ()
      call msg_fatal (" Option '" // char (option) // "' needs a value")
    end select
  end if
end function get_option_value

```



```

        string = trim (arg_value)
    end if
end function get_option_value

subroutine print_version ()
    print "(A)", "WHIZARD " // WHIZARD_VERSION
    print "(A)", "Copyright (C) 1999-2013 Wolfgang Kilian, Thorsten Ohl, Juergen Reuter,"
    print "(A)", "                Christian Speckner"
    print "(A)", "This is free software; see the source for copying conditions. There is NO"
    print "(A)", "warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE."
    print *
end subroutine print_version

subroutine print_usage ()
    print "(A)", "WHIZARD " // WHIZARD_VERSION
    print "(A)", "Usage: whizard [OPTIONS] [FILE]"
    print "(A)", "Run WHIZARD with the command list taken from FILE(s)"
    print "(A)", "Options for resetting default directories and tools" &
        // "(GNU naming conventions):"
    print "(A)", "    --prefix DIR"
    print "(A)", "    --exec_prefix DIR"
    print "(A)", "    --bindir DIR"
    print "(A)", "    --libdir DIR"
    print "(A)", "    --includedir DIR"
    print "(A)", "    --datarootdir DIR"
    print "(A)", "    --libtool LOCAL_LIBTOOL"
    print "(A)", "    --lhapdfdir DIR    (PDF sets directory)"
    print "(A)", "Other options:"
    print "(A)", "-h, --help            display this help and exit"
    print "(A)", "    --banner            display banner at startup (default)"
    print "(A)", "-e, --execute CMDS      execute SINDARIN CMDS before reading FILE(s)"
    print "(A)", "-i, --interactive       run interactively after reading FILE(s)"
    print "(A)", "-l, --library           preload process library NAME"
    print "(A)", "    --localprefix DIR"
    print "(A)", "search in DIR for local models (default: ~/.whizard)"
    print "(A)", "-L, --logfile FILE      write log to FILE (default: 'whizard.log'"
    print "(A)", "    --logging           switch on logging at startup (default)"
    print "(A)", "-m, --model NAME        preload model NAME (default: 'SM')"
    print "(A)", "    --no-banner         do not display banner at startup"
    print "(A)", "    --no-library        do not preload process library"
    print "(A)", "    --no-logfile        do not write a logfile"
    print "(A)", "    --no-logging        switch off logging at startup"
    print "(A)", "    --no-model          do not preload a model"
    print "(A)", "    --no-rebuild        do not force rebuilding"
    print "(A)", "-r, --rebuild           rebuild all (see below)"
    print "(A)", "    --rebuild-library"
    print "(A)", "rebuild process code library"
    print "(A)", "    --rebuild-user      rebuild user-provided code"
    print "(A)", "    --rebuild-phase-space"
    print "(A)", "rebuild phase-space configuration"
    print "(A)", "    --rebuild-grids     rebuild integration grids"
    print "(A)", "    --rebuild-events    rebuild event samples"
    print "(A)", "    --recompile         recompile process code, ignoring any existing library"
    print "(A)", "-u --user              enable user-provided code"

```

```

        print "(A)", "      --user-src FILE    user-provided source file"
        print "(A)", "      --user-lib FILE    user-provided library file"
        print "(A)", "      --user-target BN  basename of created user library (default: user)"
        print "(A)", "-V, --version          output version information and exit"
        print "(A)", "      --write-syntax-tables"
        print "(A)", "                                write the internal syntax tables to files and exit"
        print "(A)", "-                                further options are taken as filenames"
        print *
        print "(A)", "With no FILE, read standard input."
    end subroutine print_usage

end program main

```

## 20.5 Shower

(shower\_interface.f90)≡

*⟨File header⟩*

module shower\_interface

```

    use kinds, only: default, double !NODEP!
    use shower_basics_module !NODEP!
    use shower_module !NODEP!
    use shower_topythia_module !NODEP!
    use multi, multi_output_unit => output_unit !NODEP!
    use mlm_matching_module !NODEP!
    use ckkw_matching_module !NODEP!
    use tao_random_numbers !NODEP!
    use flavors
    use colors
    use particles
    use subevents
    use models
    use variables
    use iso_varying_string, string_t => varying_string !NODEP!

```

*⟨Use file utils⟩*

```

    use event_formats
    use os_interface
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use strfun
    use pdf_builtin !NODEP!

```

*⟨Standard module head⟩*

*⟨Shower: public⟩*

*⟨Shower: types⟩*

contains

*⟨Shower: procedures⟩*

```

end module shower_interface

<Shower: public>≡
  public :: shower_settings_t

<Shower: types>≡
  type :: shower_settings_t
    logical :: ps_isr_active = .false.
    logical :: ps_fsr_active = .false.
    logical :: ps_use_PYTHIA_shower = .false.
    logical :: hadronization_active = .false.
    logical :: mlm_matching = .false.
    logical :: ckkw_matching = .false.
    logical :: muli_active = .false.

    logical :: ps_PYTHIA_verbose = .false.
    type(string_t) :: ps_PYTHIA_PYGIVE

    !!! values present in PYTHIA and WHIZARDS PS,
    !!! comments denote corresponding PYTHIA values
    real(default) :: ps_mass_cutoff = 1._default      ! PARJ(82)
    real(default) :: ps_fsr_lambda = 0.29_default     ! PARP(72)
    real(default) :: ps_isr_lambda = 0.29_default     ! PARP(61)
    integer :: ps_max_n_flavors = 5                   ! MSTJ(45)
    logical :: ps_isr_alpha_s_running = .true.        ! MSTP(64)
    logical :: ps_fsr_alpha_s_running = .true.        ! MSTJ(44)
    real(default) :: ps_fixed_alpha_s = 0._default    ! PARU(111)
    logical :: ps_isr_pt_ordered = .false.
    logical :: ps_isr_angular_ordered = .true.        ! MSTP(62)
    real(default) :: ps_isr_primordial_kt_width = 0._default ! PARP(91)
    real(default) :: ps_isr_primordial_kt_cutoff = 5._default ! PARP(93)
    real(default) :: ps_isr_z_cutoff = 0.999_default ! 1-PARP(66)
    real(default) :: ps_isr_minenergy = 2._default   ! PARP(65)
    real(default) :: ps_isr_tscalefactor = 1._default
    logical :: ps_isr_only_onshell_emitted_partons = .true. ! MSTP(63)

    !!! MLM settings
    type(mlm_matching_settings_t) :: ms

    !!! CKKW Matching
    type(ckkw_matching_settings_t) :: ckkw_settings
    type(ckkw_pseudo_shower_weights_t) :: ckkw_weights
end type shower_settings_t

```

Read in the shower, matching and hadronization settings

```

<Shower: public>+≡
  public :: shower_settings_init

<Shower: procedures>≡
  subroutine shower_settings_init(shower_settings, var_list)
    type(shower_settings_t), intent(out) :: shower_settings
    type(var_list_t), intent(in) :: var_list

    shower_settings%ps_isr_active = &
      var_list_get_lval(var_list, var_str("?ps_isr_active"))

```

```

shower_settings%ps_fsr_active = &
    var_list_get_lval(var_list, var_str("?ps_fsr_active"))
shower_settings%hadronization_active = &
    var_list_get_lval(var_list, var_str("?hadronization_active"))
shower_settings%mlm_matching = &
    var_list_get_lval(var_list, var_str("?mlm_matching"))
shower_settings%ckkw_matching = &
    var_list_get_lval(var_list, var_str("?ckkw_matching"))
shower_settings%multi_active = &
    var_list_get_lval(var_list, var_str("?multi_active"))

!   if( (shower_settings%ps_fsr_active .eqv. .false.) .and. (shower_settings%ps_isr_active .eqv. .fal
!       .and. (shower_settings%hadronization_active .eqv. .false.) .and. (shower_settings%mlm_matchin
!       return
!   end if

shower_settings%ps_use_PYTHIA_shower = &
    var_list_get_lval(var_list, var_str("?ps_use_PYTHIA_shower"))
shower_settings%ps_PYTHIA_verbose = &
    var_list_get_lval(var_list, var_str("?ps_PYTHIA_verbose"))
shower_settings%ps_PYTHIA_PYGIVE = &
    var_list_get_sval(var_list, var_str("$ps_PYTHIA_PYGIVE"))
shower_settings%ps_mass_cutoff = &
    var_list_get_rval(var_list, var_str("ps_mass_cutoff"))
shower_settings%ps_fsr_lambda = &
    var_list_get_rval(var_list, var_str("ps_fsr_lambda"))
shower_settings%ps_isr_lambda = &
    var_list_get_rval(var_list, var_str("ps_isr_lambda"))
shower_settings%ps_max_n_flavors = &
    var_list_get_ival(var_list, var_str("ps_max_n_flavors"))
shower_settings%ps_isr_alpha_s_running = &
    var_list_get_lval(var_list, var_str("?ps_isr_alpha_s_running"))
shower_settings%ps_fsr_alpha_s_running = &
    var_list_get_lval(var_list, var_str("?ps_fsr_alpha_s_running"))
shower_settings%ps_fixed_alpha_s = &
    var_list_get_rval(var_list, var_str("ps_fixed_alpha_s"))
shower_settings%ps_isr_pt_ordered = &
    var_list_get_lval(var_list, var_str("?ps_isr_pt_ordered"))
shower_settings%ps_isr_angular_ordered = &
    var_list_get_lval(var_list, var_str("?ps_isr_angular_ordered"))
shower_settings%ps_isr_primordial_kt_width = &
    var_list_get_rval(var_list, var_str("ps_isr_primordial_kt_width"))
shower_settings%ps_isr_primordial_kt_cutoff = &
    var_list_get_rval(var_list, var_str("ps_isr_primordial_kt_cutoff"))
shower_settings%ps_isr_z_cutoff = &
    var_list_get_rval(var_list, var_str("ps_isr_z_cutoff"))
shower_settings%ps_isr_minenergy = &
    var_list_get_rval(var_list, var_str("ps_isr_minenergy"))
shower_settings%ps_isr_tscalefactor = &
    var_list_get_rval(var_list, var_str("ps_isr_tscalefactor"))
shower_settings%ps_isr_only_onshell_emitted_partons = &
    var_list_get_lval(var_list, var_str("?ps_isr_only_onshell_emitted_partons"))

!!! MLM matching

```

```

shower_settings%ms%mlm_Qcut_ME = var_list_get_rval(var_list, var_str("mlm_Qcut_ME"))
shower_settings%ms%mlm_Qcut_PS = var_list_get_rval(var_list, var_str("mlm_Qcut_PS"))
shower_settings%ms%mlm_ptmin = var_list_get_rval(var_list, var_str("mlm_ptmin"))
shower_settings%ms%mlm_etamax = var_list_get_rval(var_list, var_str("mlm_etamax"))
shower_settings%ms%mlm_Rmin = var_list_get_rval(var_list, var_str("mlm_Rmin"))
shower_settings%ms%mlm_Emin = var_list_get_rval(var_list, var_str("mlm_Emin"))
shower_settings%ms%mlm_nmaxMEjets = var_list_get_ival(var_list, var_str("mlm_nmaxMEjets"))

shower_settings%ms%mlm_ETclusfactor = var_list_get_rval(var_list, var_str("mlm_ETclusfactor"))
shower_settings%ms%mlm_ETclusminE = var_list_get_rval(var_list, var_str("mlm_ETclusminE"))
shower_settings%ms%mlm_etaclusfactor = var_list_get_rval(var_list, var_str("mlm_etaclusfactor"))
shower_settings%ms%mlm_Rclusfactor = var_list_get_rval(var_list, var_str("mlm_Rclusfactor"))
shower_settings%ms%mlm_Eclusfactor = var_list_get_rval(var_list, var_str("mlm_Eclusfactor"))

!!! CKKW matching
! TODO
end subroutine shower_settings_init

```

*<Shower: public>+≡*

```
public :: shower_settings_write
```

*<Shower: procedures>+≡*

```

subroutine shower_settings_write(shower_settings, unit)
  type(shower_settings_t), intent(in) :: shower_settings
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit(unit); if (u < 0) return
  write (u, "(A)") "Shower Settings:"
  write (u, *) "ps_isr_active" = ", shower_settings%ps_isr_active"
  write (u, *) "ps_fsr_active" = ", shower_settings%ps_fsr_active"
  if (shower_settings%ps_isr_active .or. shower_settings%ps_fsr_active) then
    write (u, *) "ps_use_PYTHIA_shower" = ", shower_settings%ps_use_PYTHIA_shower"
    write (u, *) "ps_mass_cutoff" = ", shower_settings%ps_mass_cutoff"
    write (u, *) "ps_max_n_flavors" = ", shower_settings%ps_max_n_flavors"
  end if
  if (shower_settings%ps_isr_active) then
    write (u, "(A)") " ISR Settings:"
    write (u, *) "ps_isr_pt_ordered" = ", shower_settings%ps_isr_pt_ordered"
    write (u, *) "ps_isr_lambda" = ", shower_settings%ps_isr_lambda"
    write (u, *) "ps_isr_alpha_s_running" = ", shower_settings%ps_isr_alpha_s_running"
    write (u, *) "ps_isr_primordial_kt_width" = ", shower_settings%ps_isr_primordial_kt_width"
    write (u, *) "ps_isr_primordial_kt_cutoff" = ", shower_settings%ps_isr_primordial_kt_cutoff"
    write (u, *) "ps_isr_z_cutoff" = ", shower_settings%ps_isr_z_cutoff"
    write (u, *) "ps_isr_minenergy" = ", shower_settings%ps_isr_minenergy"
    write (u, *) "ps_isr_tscalefactor" = ", shower_settings%ps_isr_tscalefactor"
  end if
  if (shower_settings%ps_fsr_active) then
    write (u, "(A)") " FSR Settings:"
    write (u, *) "ps_fsr_lambda" = ", shower_settings%ps_fsr_lambda"
    write (u, *) "ps_fsr_alpha_s_running" = ", shower_settings%ps_fsr_alpha_s_running"
  end if
  write (u, "(A)") "Hadronization Settings:"
  write (u, *) "hadronization_active" = ", shower_settings%hadronization_active"
  write (u, "(A)") "Matching Settings:"

```

```

write (u, *) "mlm_matching          = ", shower_settings%mlm_matching
if (shower_settings%mlm_matching) then
  call mlm_matching_settings_write(shower_settings%ms, u)
end if
write (u, *) "ckkw_matching          = ", shower_settings%ckkw_matching
if (shower_settings%ckkw_matching) then
  ! TODO ckkw settings etc.
end if
write (u, *)
write (u, *) "ps_PYTHIA_verbose          = ", shower_settings%ps_PYTHIA_verbose
write (u, *) "ps_PYTHIA_PYGIVE          = ", char(shower_settings%ps_PYTHIA_PYGIVE)
end subroutine shower_settings_write

```

The wrapper subroutine for the shower and the hadronization interface and MLM matching. This should be the only subroutine called from WHIZARD.

*<Shower: public>+≡*

```
public :: apply_shower_particle_set
```

*<Shower: procedures>+≡*

```

subroutine apply_shower_particle_set(particle_set, shower_settings, model, &
  os_data, pdf_type, pdf_set, valid, vetoed)
  type(particle_set_t), intent(inout) :: particle_set
  type(shower_settings_t), intent(in) :: shower_settings
  type(model_t), pointer, intent(in) :: model
  type(os_data_t), intent(in) :: os_data
  integer, intent(in) :: pdf_type
  integer, intent(in) :: pdf_set
  logical, intent(inout) :: valid
  logical, intent(inout) :: vetoed
  real(kind=double) :: pdftest

  type(mlm_matching_data_t) :: mlm_matching_data
  logical, save :: matching_disabled=.false.
  procedure(shower_pdf), pointer :: pdf_func => null()

  interface
    subroutine evolvePDFM (set, x, q, ff)
      integer, intent(in) :: set
      double precision, intent(in) :: x, q
      double precision, dimension(-6:6), intent(out) :: ff
    end subroutine evolvePDFM
  end interface

  if ((shower_settings%ps_fsr_active .eqv. .false.) .and. &
    (shower_settings%ps_isr_active.eqv..false.) .and. &
    (shower_settings%hadronization_active.eqv..false.) .and. &
    (shower_settings%mlm_matching.eqv..false.) ) then
    ! return if nothing to do
    return
  end if

  ! return if already invalid or vetoed
  if ((.not.valid) .or. vetoed) then
    return
  end if

```

```

end if
! ensure that lhpdf is initialized
if (pdf_type .eq. STRF_LHAPDF) then
  if (shower_settings%ps_isr_active .and. &
      (abs (particle_get_pdg (particle_set_get_particle (particle_set, 1))).ge.1000) .and. &
      (abs (particle_get_pdg (particle_set_get_particle (particle_set, 2))).ge.1000)) then
    call GetQ2max (0,pdftest)
    if (pdftest .eq. 0._double) then
      call msg_fatal (" ISR enabled, but LHAPDF not initialized," // &
                     char(10) // "      aborting simulation")
      return
    end if
  end if
  pdf_func => evolvePDFM
else if (pdf_type.eq. STRF_PDF_BUILTIN) then
  if (shower_settings%ps_use_PYTHIA_shower) then
    call msg_fatal (" builtin-pdfs can not be used for PYTHIA showers," // &
                   char(10) // "      aborting simulation")
    return
  end if
  pdf_func => pdf_evolve_LHAPDF
end if
if (shower_settings%mlm_matching .and. shower_settings%ckkw_matching) then
  call msg_fatal (" both MLM and CKKW matching activated," // &
                 char(10) // "      aborting simulation")
  return
end if

!   call shower_settings_write(shower_settings)

if(shower_settings%ps_PYTHIA_verbose.eqv..false.) then
  call PYGIVE('MSTU(12)=12345')
  call PYGIVE('MSTU(13)=0')
else
  call PYGIVE('MSTU(13)=1')
end if
!   pause

if(matching_disabled.eqv..false.) then
  !!! Check if the beams are hadrons
  if ((abs (particle_get_pdg(particle_set_get_particle (particle_set, 1))).le.18) .and. &
      (abs (particle_get_pdg(particle_set_get_particle (particle_set, 2))).le.18)) then
    mlm_matching_data%is_hadron_collision = .false.
  else if ((abs(particle_get_pdg (particle_set_get_particle (particle_set, 1))).ge.1000) .and.
          (abs(particle_get_pdg (particle_set_get_particle (particle_set, 2))).ge.1000)) then
    mlm_matching_data%is_hadron_collision = .true.
  else
    call msg_error (" Matching didn't recognize beams setup," // &
                   char(10) // "      disabling matching")
    matching_disabled = .true.
    return
  end if
end if
end if

```

```

!!! SHOWER
  if (shower_settings%ps_use_PYTHIA_shower .or. &
      ((.not.shower_settings%ps_fsr_active) .and. (.not.shower_settings%ps_isr_active) &
       .and.(shower_settings%hadronization_active))) then
    call apply_PYTHIAshower_particle_set (particle_set, shower_settings, mlm_matching_data%P_ME
    !      call pylist(2)
  else
    call apply_WHIZARDshower_particle_set (particle_set, shower_settings, mlm_matching_data%P_M
      os_data, pdf_func, pdf_set, valid, vetoed)
    if(vetoed) return
  end if
  !call particle_set_write(particle_set)
  !print *, " after SHOWER"
  !pause

  if(shower_settings%mlm_matching.and.(matching_disabled.eqv..false.)) then
!!! MLM stage 2 -> PS jets and momenta
    call matching_transfer_PS (mlm_matching_data, particle_set, shower_settings)
!!! MLM stage 3 -> reconstruct and possible reject
    call mlm_matching (mlm_matching_data, shower_settings%ms, vetoed)
    if (vetoed) then
      call mlm_matching_data_final (mlm_matching_data)
      return
    end if
  endif

!!! HADRONIZATION
  if (shower_settings%hadronization_active) then
    !! Assume that the event record is still in the PYTHIA COMMON BLOCKS
    !! transferred there by one of the shower routines
    if (valid) then
      call apply_PYTHIAhadronization (particle_set, shower_settings, model, valid)
    end if
  end if

!!! FINAL

  call mlm_matching_data_final (mlm_matching_data)
!   print *, "SHOWER+HAD+MATCHING finished"
contains
  subroutine shower_set_PYTHIA_error (mstu23)
    ! PYTHIA common blocks
    IMPLICIT DOUBLE PRECISION(A-H, O-Z)
    IMPLICIT INTEGER(I-N)
    COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
    SAVE/PYDAT1/

    integer, intent(in) :: mstu23

    MSTU(23) = mstu23
  end subroutine shower_set_PYTHIA_error

  function shower_get_PYTHIA_error () result (mstu23)
    ! PYTHIA common blocks
    IMPLICIT DOUBLE PRECISION(A-H, O-Z)

```



```

      IMPLICIT INTEGER(I-N)
      COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
      SAVE/PYDAT1/

      integer :: mstu23

      mstu23 = MSTU(23)
end function shower_get_PYTHIA_error

      subroutine apply_PYTHIAshower_particle_set (particle_set, shower_settings, JETS_ME, model, val
      <HEP common: parameters>
      <HEP common: variables>
      <HEP common: common blocks>

      type(particle_set_t), intent(inout) :: particle_set
      type(particle_set_t) :: pset_reduced
      type(shower_settings_t), intent(in) :: shower_settings
      type(vector4_t), dimension(:), allocatable, intent(inout) :: JETS_ME
      type(model_t), pointer, intent(in) :: model
      logical, intent(inout) :: valid
      real(kind=default) :: rand

      ! units for transfer from WHIZARD to PYTHIA and back
      integer :: u_W2P, u_P2W
      integer, save :: pythia_initialized_for_NPRUP = 0
      logical, save :: pythia_warning_given = .false.
      logical, save :: msg_written = .false.
      type(string_t) :: remaining_PYGIVE, partial_PYGIVE
      character*10 buffer

      ! check if the beam particles are quarks
      if ((abs(IDBMUP(1)).le.8).or.(abs(IDBMUP(2)).le.8)) then
        ! PYTHIA doesn't support these settings
        if (pythia_warning_given.eqv..false.) then
          call msg_error ("PYTHIA doesn't support quarks as beam particles," // &
            char(10) // "      negelecting ISR, FSR and hadronization")
          pythia_warning_given = .true.
        end if
        return
      end if

      call particle_set_reduce (particle_set, pset_reduced)
      call particle_set_fill_hepeup (pset_reduced)
      call hepeup_set_event_parameters (proc_id=1)

      call shower_set_pythia_error(0)
      u_W2P = free_unit()
      ! only for debugging purposes
      ! open (unit=u_W2P, status="replace", file="whizardout1.lhe", action="readwrite")
      open (unit=u_W2P, status="scratch", action="readwrite")
      call les_houches_events_write_header (u_W2P)
      call heprup_write_lhef (u_W2P)
      call hepeup_write_lhef (u_W2P)
      call les_houches_events_write_footer (u_W2P)

```

```

rewind (u_W2P)
write (buffer, "(I10)") u_W2P
call pygive ("MSTP(161)="//buffer)
call pygive ("MSTP(162)="//buffer)
if (shower_settings%ps_isr_active.eqv..false.) then
  call pygive ("MSTP(61)=0") ! switch off ISR
else
  call pygive ("MSTP(61)=1")
end if
if (shower_settings%ps_fsr_active.eqv..false.) then
  call pygive ("MSTP(71)=0") ! switch off FSR
else
  call pygive ("MSTP(71)=1")
end if
call pygive ("MSTP(111)=0") ! switch off hadronization

if (pythia_initialized_for_NPRUP .ge. NPRUP) then
  call upinit
else
  write (buffer, "(F10.5)") shower_settings%ps_mass_cutoff
  call pygive ("PARJ(82)="//buffer)
  write (buffer, "(F10.5)") shower_settings%ps_isr_tscalefactor
  call pygive ("PARP(71)="//buffer)

  write (buffer, "(F10.5)") shower_settings%ps_fsr_lambda
  call pygive ("PARP(72)="//buffer)
  write (buffer, "(F10.5)") shower_settings%ps_isr_lambda
  call pygive ("PARP(61)="//buffer)
  write (buffer, "(I10)") shower_settings%ps_max_n_flavors
  call pygive ("MSTJ(45)="//buffer)
  if (shower_settings%ps_isr_alpha_s_running) then
    call pygive ("MSTP(64)=2")
  else
    call pygive ("MSTP(64)=0")
  end if
  if (shower_settings%ps_fsr_alpha_s_running) then
    call pygive ("MSTJ(44)=2")
  else
    call pygive ("MSTJ(44)=0")
  end if
  write (buffer, "(F10.5)") shower_settings%ps_fixed_alpha_s
  call pygive ("PARU(111)="//buffer)
  write (buffer, "(F10.5)") shower_settings%ps_isr_primordial_kt_width
  call pygive ("PARP(91)="//buffer)
  write (buffer, "(F10.5)") shower_settings%ps_isr_primordial_kt_cutoff
  call pygive ("PARP(93)="//buffer)
  write (buffer, "(F10.5)") 1._double - shower_settings%ps_isr_z_cutoff
  call pygive ("PARP(66)="//buffer)
  write (buffer, "(F10.5)") shower_settings%ps_isr_minenergy
  call pygive ("PARP(65)="//buffer)
  if (shower_settings%ps_isr_only_onshell_emitted_partons) then
    call pygive ("MSTP(63)=0")
  else
    call pygive ("MSTP(63)=2")
  end if
end if

```

```

end if
if (shower_settings%mlm_matching) then
  CALL PYGIVE('MSTP(62)=2')
  CALL PYGIVE('MSTP(67)=0')
end if
call pyinit ("USER", "", "", 0D0)

call tao_random_number (rand)
write (buffer, "(I10)") floor (rand*900000000)
call pygive ("MRPY(1)="//buffer)
call pygive ("MRPY(2)=0")

if (len(shower_settings%ps_PYTHIA_PYGIVE)>0) then
  remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE
  do while (len (remaining_PYGIVE)>0)
    call split (remaining_PYGIVE, partial_PYGIVE, ";")
    call PYGIVE (char (partial_PYGIVE))
  end do
  if (shower_get_PYTHIA_error().ne.0) then
    call msg_fatal (" PYTHIA didn't recognize ps_PYTHIA_PYGIVE setting")
  end if
end if

pythia_initialized_for_NPRUP = NPRUP
end if

if (.not.msg_written) then
  call msg_message ("Using PYTHIA interface for parton showers")
  msg_written = .true.
end if
call pyevnt ()

u_P2W = free_unit ()
write (buffer, "(I10)") u_P2W
call pygive ("MSTP(163)="//buffer)
!open(unit=u_P2W, file="pythiaout.lhe", status="replace", action="readwrite") ! only for de
open(unit=u_P2W, status="scratch", action="readwrite")
! convert pythia /PYJETS/ to lhef given in MSTU(163)=u_P2W
call pylheo
! read and add lhef from u_P2W
call shower_add_lhef_to_particle_set(particle_set, u_P2W, model, os_data)
close(unit=u_P2W)

if(shower_settings%mlm_matching) then
  ! transfer momenta of the partons in the final state of the hard ininteraction
  call get_ME_momenta_from_PYTHIA(JETS_ME)
end if
close(unit=u_W2P)
valid = (shower_get_PYTHIA_error().eq.0)
end subroutine apply_PYTHIAshower_particle_set
subroutine apply_WHIZARDshower_particle_set(particle_set, shower_settings, JETS_ME, model_in,
  os_data, pdf_func, pdf_set, valid, vetoed)
  type(particle_set_t), intent(inout) :: particle_set
  type(shower_settings_t), intent(in) :: shower_settings

```

```

type(vector4_t), dimension(:), allocatable, intent(inout) :: JETS_ME
type(model_t), pointer, intent(in) :: model_in
type(os_data_t), intent(in) :: os_data
procedure(shower_pdf), pointer, intent(in) :: pdf_func
integer, intent(in) :: pdf_set
logical, intent(inout) :: valid
logical, intent(out) :: vetoed

type(muli_type), save :: mi
type(shower_t) :: shower
type(parton_t), dimension(:), allocatable, target :: partons, hadrons
type(parton_pointer_t), dimension(:), allocatable :: parton_pointers, final_ME_partons
real(kind=default) :: mi_scale, ps_scale, shat, phi
type(parton_pointer_t) :: temppp
integer, dimension(:), allocatable :: connections
integer :: n_loop, i, j, k
integer :: n_hadrons, n_in, n_out
integer :: n_interactions
integer :: max_color_nr
integer, dimension(2) :: col_array
integer, dimension(1) :: parent
type(flavor_t) :: flv
type(color_t) :: col
type(model_t), pointer :: model
type(model_t), target, save :: model_SM_hadrons
logical, save :: model_SM_hadrons_associated = .false.
logical, save :: msg_written = .false.
logical :: exist_SM_hadrons
type(string_t) :: filename
integer, dimension(2,4) :: color_corr
integer :: color_i, color_j
character*5 buffer
integer :: u_S2W

vetoed = .false.

! transfer settings from shower_settings to shower
call shower_set_D_Min_t (shower_settings%ps_mass_cutoff**2)
call shower_set_D_Lambda_fsr (shower_settings%ps_fsr_lambda)
call shower_set_D_Lambda_isr (shower_settings%ps_isr_lambda)
call shower_set_D_Nf (shower_settings%ps_max_n_flavors)
call shower_set_D_running_alpha_s_fsr (shower_settings%ps_fsr_alpha_s_running)
call shower_set_D_running_alpha_s_isr (shower_settings%ps_isr_alpha_s_running)
call shower_set_D_constantalpha_s (shower_settings%ps_fixed_alpha_s)
call shower_set_isr_pt_ordered (shower_settings%ps_isr_pt_ordered)
call shower_set_isr_angular_ordered (shower_settings%ps_isr_angular_ordered)
call shower_set_primordial_kt_width (shower_settings%ps_isr_primordial_kt_width)
call shower_set_primordial_kt_cutoff (shower_settings%ps_isr_primordial_kt_cutoff)
call shower_set_maxz_isr (shower_settings%ps_isr_z_cutoff)
call shower_set_minenergy_timelike (shower_settings%ps_isr_minenergy)
call shower_set_tscalefactor_isr (shower_settings%ps_isr_tscalefactor)
call shower_set_isr_only_onshell_emitted_partons &
    (shower_settings%ps_isr_only_onshell_emitted_partons)
call shower_set_pdf_set (pdf_set)

```

```

call shower_set_pdf_func (pdf_func)

if (.not.msg_written) then
  call msg_message ("Using WHIZARD's internal showering")
  msg_written = .true.
end if

n_loop = 0
try_shower: do ! just a loop to be able to discard events
  n_loop = n_loop + 1
  if (n_loop .gt. 1000) STOP "BUG: too many loops (try_shower)"
  call shower_create (shower)
  max_color_nr = 0

  n_hadrons = 0
  n_in = 0
  n_out = 0
  do i = 1, particle_set_get_n_tot (particle_set)
    if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_BEAM) &
      n_hadrons = n_hadrons+1
    if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_INCOMING)
      n_in = n_in+1
    if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_OUTGOING)
      n_out = n_out+1
    end do

    allocate (connections (1:particle_set_get_n_tot (particle_set)))
    connections = 0

    allocate (hadrons (1:2))
    allocate (partons (1:n_in+n_out))
    allocate (parton_pointers (1:n_in+n_out))

    j=0
    if (n_hadrons > 0) then
      ! Transfer hadrons
      do i = 1, particle_set_get_n_tot (particle_set)
        if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_BEAM)
          j = j+1
          hadrons(j)%nr = shower_get_next_free_nr (shower)
          hadrons(j)%momentum = particle_get_momentum (particle_set_get_particle (particle_set, i))
          hadrons(j)%t = hadrons(j)%momentum**2
          hadrons(j)%typ = particle_get_pdg (particle_set_get_particle (particle_set, i))
          col_array=particle_get_color (particle_set_get_particle (particle_set, i))
          hadrons(j)%c1 = col_array(1)
          hadrons(j)%c2 = col_array(2)
          max_color_nr = max (max_color_nr, abs(hadrons(j)%c1), abs(hadrons(j)%c2))
          hadrons(j)%interactionnr = 1
          connections(i)=j
        end if
      end do
    end if

    ! transfer incoming partons

```

```

j = 0
do i = 1, particle_set_get_n_tot (particle_set)
  if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_INCOMING)
    j = j+1
    partons(j)%nr = shower_get_next_free_nr (shower)
    partons(j)%momentum = particle_get_momentum (particle_set_get_particle (particle_set, i))
    partons(j)%t = partons(j)%momentum**2
    partons(j)%typ = particle_get_pdg (particle_set_get_particle (particle_set, i))
    col_array=particle_get_color (particle_set_get_particle (particle_set, i))
    partons(j)%c1 = col_array (1)
    partons(j)%c2 = col_array (2)
    parton_pointers(j)%p => partons(j)
    max_color_nr = max (max_color_nr, abs (partons(j)%c1), abs (partons(j)%c2))
    connections(i)=j
    ! insert dependences on hadrons
    if (particle_get_n_parents (particle_set_get_particle (particle_set, i))==1) then
      parent = particle_get_parents (particle_set_get_particle (particle_set, i))
      partons(j)%initial => hadrons (connections (parent(1)))
      partons(j)%x = space_part_norm (partons(j)%momentum) / &
        space_part_norm (partons(j)%initial%momentum)
    end if
  end if
end do
! transfer outgoing partons
do i = 1, particle_set_get_n_tot (particle_set)
  if (particle_get_status (particle_set_get_particle (particle_set, i)) == PRT_OUTGOING)
    j = j+1
    partons(j)%nr = shower_get_next_free_nr (shower)
    partons(j)%momentum = particle_get_momentum (particle_set_get_particle (particle_set, i))
    partons(j)%t = partons(j)%momentum**2
    partons(j)%typ = particle_get_pdg (particle_set_get_particle (particle_set, i))
    col_array=particle_get_color (particle_set_get_particle (particle_set, i))
    partons(j)%c1 = col_array(1)
    partons(j)%c2 = col_array(2)
    parton_pointers(j)%p => partons(j)
    max_color_nr = max (max_color_nr, abs (partons(j)%c1), abs (partons(j)%c2))
    connections(i)=j
  end if
end do

deallocate(connections)

! insert these partons in shower
call shower_set_next_color_nr (shower, 1+max_color_nr)
call shower_add_interaction2tonCKKW (shower, parton_pointers, shower_settings%ckkw_weight)

if (shower_settings%multi_active) then
  ! Initialize multi pdf sets, unless initialized
  if (mi%is_initialized ()) then
    call mi%restart ()
  else
!!$      call shower_print(shower)
!!$      print *, "-----"

```

```

!!$          call interaction_print(shower%interactions(i)%i)
!!$          print *, "-----"
!!$          call vector4_write(shower%interactions(1)%i%partons(1)%p%momentum)
!!$          call vector4_write(shower%interactions(1)%i%partons(2)%p%momentum)
!!$          print *, "-----"
call mi%initialize (&
    GeV2_scale_cutoff=D_Min_t, &
    GeV2_s=shower_interaction_get_s (shower%interactions(1)%i), &
    muli_dir=char(os_data%whizard_mulipath))
end if

! initial interaction
call mi%apply_initial_interaction(&
    GeV2_s=shower_interaction_get_s(shower%interactions(1)%i), &
    x1=shower%interactions(1)%i%partons(1)%p%parent%x, &
    x2=shower%interactions(1)%i%partons(2)%p%parent%x, &
    pdg_f1=shower%interactions(1)%i%partons(1)%p%parent%typ, &
    pdg_f2=shower%interactions(1)%i%partons(2)%p%parent%typ, &
    n1=shower%interactions(1)%i%partons(1)%p%parent%nr, &
    n2=shower%interactions(1)%i%partons(2)%p%parent%nr)
end if

if (shower_settings%ckkw_matching) then
    ! CKKW Matching
    call ckkw_matching (shower, shower_settings%ckkw_settings, shower_settings%ckkw_weight)
    if (vetoed) then
        return
    end if
end if

if (shower_settings%ps_isr_active) then
    i = 0
    branchings: do
        i = i+1
        if (shower_settings%mul_i_active) then
            call mi%generate_gev2_pt2 (shower_get_ISR_scale(shower), mi_scale)
        else
            mi_scale = 0.0
        end if

        ! shower_generate_next_isr_branching returns a pointer to the parton with the next
        !           temppp=shower_generate_next_isr_branching_veto(shower)
        temppp=shower_generate_next_isr_branching(shower)

        if((.not. associated(temppp%p)).and.(mi_scale.lt.D_Min_t)) then
            exit branchings
        end if
        ! check if branching or interaction occurs next
        if(associated(temppp%p)) then
            ps_scale = abs(temppp%p%t)
        else
            ps_scale = 0._default
        end if
        if(mi_scale .gt. ps_scale) then

```

```

! discard branching evolution lower than mi_scale
call shower_set_max_ISR_scale(shower, mi_scale)
if(associated(temppp%p)) call parton_set_simulated(temppp%p, .false.)

! execute new interaction
deallocate(partons)
deallocate(parton_pointers)
allocate(partons(1:4))
allocate(parton_pointers(1:4))
do j=1,4
    partons(j)%nr = shower_get_next_free_nr(shower)
    partons(j)%belongstointeraction = .true.
    parton_pointers(j)%p => partons(j)
end do
call mi%generate_partons(partons(1)%nr, partons(2)%nr, &
    partons(1)%x, partons(2)%x, &
    partons(1)%typ, partons(2)%typ, &
    partons(3)%typ, partons(4)%typ)
! calculate momenta
shat = partons(1)%x*partons(2)%x*shower_interaction_get_s(shower%interactions(1)
call parton_set_momentum(partons(1), 0.5_default*sqrt(shat), 0._default, 0._defa
call parton_set_momentum(partons(2), 0.5_default*sqrt(shat), 0._default, 0._defa
call parton_set_initial(partons(1), shower%interactions(1)%i%partons(1)%p%initia
call parton_set_initial(partons(2), shower%interactions(1)%i%partons(2)%p%initia
partons(1)%belongstoFSR=.false.
partons(2)%belongstoFSR=.false.
! calculate color connections
call mi%get_color_correlations(shower_get_next_color_nr(shower),max_color_nr,col
call shower_set_next_color_nr(shower, max_color_nr)

partons(1)%c1 = color_corr(1,1)
partons(1)%c2 = color_corr(2,1)
partons(2)%c1 = color_corr(1,2)
partons(2)%c2 = color_corr(2,2)
partons(3)%c1 = color_corr(1,3)
partons(3)%c2 = color_corr(2,3)
partons(4)%c1 = color_corr(1,4)
partons(4)%c2 = color_corr(2,4)

call tao_random_number(phi)
phi = 2*pi*phi
call parton_set_momentum(partons(3), 0.5_default*sqrt(shat), sqrt(mi_scale)*cos(
    sqrt(mi_scale)*sin(phi), sqrt(0.25_default*shat - mi_scale))
call parton_set_momentum(partons(4), 0.5_default*sqrt(shat), -sqrt(mi_scale)*cos
    -sqrt(mi_scale)*sin(phi), -sqrt(0.25_default*shat - mi_scale))
partons(3)%belongstoFSR=.true.
partons(4)%belongstoFSR=.true.

call shower_add_interaction2ton(shower, parton_pointers)
n_interactions = size(shower%interactions)
do k=1,2
    call mi%replace_parton(shower%interactions(n_interactions)%i%partons(k)%p%ini
        shower%interactions(n_interactions)%i%partons(k)%p%nr, &
        shower%interactions(n_interactions)%i%partons(k)%p%parent%nr, &

```



```

        shower%interactions(n_interactions)%i%partons(k)%p%typ, &
        shower%interactions(n_interactions)%i%partons(k)%p%x, &
        mi_scale)
    end do
    call shower_print(shower)
    pause
!
else
    ! execute the next branching 'found' in the previous step
    call shower_execute_next_isr_branching(shower, temppp)
    if(shower_settings%multi_active) then
        call mi%replace_parton(temppp%p%initial%nr, temppp%p%child1%nr, temppp%p%nr,
            temppp%p%typ, temppp%p%x, ps_scale)
    end if
    !
    ! call shower_print(shower)
    ! pause
end if
end do branchings

call shower_generate_fsr_for_partons_emitted_in_ISR(shower)
! call shower_print(shower)
else
    call shower_simulate_no_isr_shower(shower)
end if

! some bookkeeping, needed after the shower is done
call shower_boost_to_labframe(shower)
call shower_generate_primordial_kt(shower)
call shower_update_beamremnants(shower)
! clean-up multi: we should finalize the multi pdf sets when all runs are done.
! call mi%finalize()

if(shower_settings%ps_fsr_active) then
    ! FSR
    do i=1, size(shower%interactions)
        call shower_interaction_generate_fsr2ton(shower, shower%interactions(i)%i)
    end do
else
    call shower_simulate_no_fsr_shower(shower)
end if
! call shower_print(shower)
! print *, "SHOWER_FINISHED"

if(shower_settings%mlm_matching) then
    ! transfer momenta of the partons in the final state of the hard interaction
    if(allocated(JETS_ME)) deallocate(JETS_ME)
    call shower_get_final_colored_ME_partons(shower, final_ME_partons)
    if(allocated(final_ME_partons)) then
        allocate(JETS_ME(1:size(final_ME_partons)))
        do i=1, size(final_ME_partons)
            ! transfer
            JETS_ME(i) = final_ME_partons(i)%p%momentum
        end do
        deallocate(final_ME_partons)
    end if
end if

```

```

end if

u_S2W = free_unit()
!!! only for debugging purposes
! open(unit=u_S2W, file="showerout.lhe", status="replace", action="readwrite")
open(unit=u_S2W, status="scratch", action="readwrite")
call shower_write_lhef(shower, u_S2W)
call shower_add_lhef_to_particle_set(particle_set, u_S2W, model_in, os_data)
close(u_S2W)

! move the particle data to the PYTHIA COMMON BLOCKS in case
! hadronization is active
if(shower_settings%hadronization_active) then
  call shower_converttopythia(shower)
end if
! deallocate(hadrons) ! hadrons are deallocated by shower_final
deallocate(partons)
deallocate(parton_pointers)
exit try_shower
end do try_shower
! call particle_set_write(particle_set)
! print *, "-----apply_shower_particle_set-----"
! print *, "-----"

! if(size(shower%interactions).ge.2) then
!   call shower_print(shower)
!   pause
! end if

call shower_final(shower)
! clean-up muli: we should finalize the muli pdf sets when all runs are done.
! call mi%finalize()
return
end subroutine apply_WHIZARDshower_particle_set

subroutine get_ME_momenta_from_PYTHIA(JETS_ME)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
  SAVE /PYJETS/

  type(vector4_t), dimension(:), allocatable :: JETS_ME
  real(kind=default), dimension(:,:), allocatable :: pdum

  integer :: i, j, n_jets

  if (allocated(JETS_ME)) deallocate(JETS_ME)
  if (allocated(pdum)) deallocate (pdum)

  ! final ME partons start in 7th row of event record
  i=7
  ! find number of jets
  n_jets=0
  do

```

```

        if(K(I,1).ne.21) exit
        if((K(I,2).eq.21).or.(ABS(K(I,2)).le.6)) then
            n_jets = n_jets +1
        end if
        i=i+1
    end do

    if(n_jets.eq.0) return
    allocate (JETS_ME(1:n_jets))
    allocate (pdum(1:n_jets,4))

    ! transfer jets
    i=7
    j=1
    pdum = p
    do
        if(K(I,1).ne.21) exit
        if((K(I,2).eq.21).or.(ABS(K(I,2)).le.6)) then
            JETS_ME(j)= vector4_moving(pdum(I,4), &
                vector3_moving( (/pdum(I,1),pdum(I,2),pdum(I,3) /) ) )
            j=j+1
        end if
        i=i+1
    end do
end subroutine get_ME_momenta_from_PYTHIA
subroutine matching_transfer_PS(mlm_matching_data, particle_set, shower_settings)
    ! transfer partons after parton shower to mlm_matching_data%P_PS
    type(mlm_matching_data_t), intent(inout) :: mlm_matching_data
    type(particle_set_t), intent(in) :: particle_set
    type(shower_settings_t), intent(in) :: shower_settings

    integer :: i, j, n_jets_PS
    integer, dimension(2) :: col
    type(particle_t) :: tempprt
    real(kind=double) :: eta, E, pl

    ! loop over particles and extract final colored ones with eta<etamax
    n_jets_PS=0
    do i=1, particle_set_get_n_tot(particle_set)
        tempprt = particle_set_get_particle(particle_set, i)
        if(particle_get_status(tempprt)/=PRT_OUTGOING) cycle
        col = particle_get_color(tempprt)
        if(all(col == 0)) cycle
        if(mlm_matching_data%is_hadron_collision) then
            E = vector4_get_component(particle_get_momentum(tempprt),0)
            pl = vector4_get_component(particle_get_momentum(tempprt),3)
            eta = 0.5*abs(log( min((E+pl)/(E-pl), 1D10)))
            if (eta> shower_settings%ms%mlm_etaClusfactor*shower_settings%ms%mlm_etamax) then
                print *, "REJECTING"
                call particle_write(tempprt)
                cycle
            end if
        end if
        n_jets_PS=n_jets_PS + 1
    end do

```

```

end do

allocate(mlm_matching_data%P_PS(1:n_jets_PS))
! print *, "n_jets_ps=", n_jets_ps

j=1
do i=1, particle_set_get_n_tot(particle_set)
  tempprt = particle_set_get_particle(particle_set, i)
  if(particle_get_status(tempprt)/=PRT_OUTGOING) cycle
  col = particle_get_color(tempprt)
  if(all(col == 0)) cycle
  if(mlm_matching_data%is_hadron_collision) then
    E = vector4_get_component(particle_get_momentum(tempprt),0)
    pl = vector4_get_component(particle_get_momentum(tempprt),3)
    eta = 0.5*abs(log( min((E+pl)/(E-pl), 1D10)))
    if (eta> shower_settings%ms/mlm_etaClusfactor*shower_settings%ms/mlm_etamax) cycle
  end if
  mlm_matching_data%P_PS(j) = particle_get_momentum(tempprt)
  j=j + 1
end do
end subroutine matching_transfer_PS

subroutine apply_PYTHIAhadronization(particle_set, shower_settings, model, valid)
  type(particle_set_t), intent(inout) :: particle_set
  type(shower_settings_t), intent(in) :: shower_settings
  type(model_t), pointer, intent(in) :: model
  logical, intent(inout) :: valid
  integer :: u_W2P, u_P2W
  type(string_t) :: remaining_PYGIVE, partial_PYGIVE
  logical, save :: msg_written = .false.
  character*10 buffer

  if(shower_settings%hadronization_active.eqv..false.) then
    return
  end if
  if(.not.valid) return

  u_W2P = free_unit()
  !open(unit=u_W2P, status="replace", file="whizardout.lhe", action="readwrite") ! only for d
  open(unit=u_W2P, status="scratch", action="readwrite")
  call les_houches_events_write_header (u_W2P)
  call heprup_write_lhef(u_W2P)
  call hepeup_write_lhef(u_W2P)
  call les_houches_events_write_footer (u_W2P)
  rewind(u_W2P)
  write (buffer, "(I10)") u_W2P
  call pygive ("MSTP(161)="//buffer)
  call pygive ("MSTP(162)="//buffer)

  ! Assume that the event is still present in the PYTHIA common blocks
! call pygive ("MSTP(61)=0") ! switch off ISR
! call pygive ("MSTP(71)=0") ! switch off FSR

  if((shower_settings%ps_use_PYTHIA_shower.eqv..false.) .and. len(shower_settings%ps_PYTHIA_PY
    remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE

```

```

do while(len(remaining_PYGIVE)>0)
  call split(remaining_PYGIVE, partial_PYGIVE, ";")
  call PYGIVE(char(partial_PYGIVE))
end do
if(shower_get_PYTHIA_error().ne.0) then
  call msg_fatal(" PYTHIA didn't recognize ps_PYTHIA_PYGIVE setting")
end if
end if

if(.not.(shower_settings%ps_use_PYTHIA_shower.and.(shower_settings%ps_isr_active.or. &
  shower_settings%ps_fsr_active))) then
  if(len(shower_settings%ps_PYTHIA_PYGIVE)>0) then
    remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE
    do while(len(remaining_PYGIVE)>0)
      call split(remaining_PYGIVE, partial_PYGIVE, ";")
      call PYGIVE(char(partial_PYGIVE))
    end do
    if(shower_get_PYTHIA_error().ne.0) then
      call msg_fatal(" PYTHIA didn't recognize ps_PYTHIA_PYGIVE setting")
    end if
  end if
end if

if(.not.msg_written) then
  call msg_message("Using PYTHIA interface for hadronization and decays")
  msg_written = .true.
end if

call pygive ("MSTP(111)=1") ! switch on hadronization
call PYEXEC

if (shower_get_PYTHIA_error() .gt. 0) then
  ! clean up, discard shower and exit
  call shower_set_PYTHIA_error(0)
  close(u_W2P)
  valid = .false.
else
  ! convert back
  u_P2W = free_unit()
  write (buffer, "(I10)") u_P2W
  call pygive ("MSTP(163)="//buffer)
  !open(unit=u_P2W, file="pythiaout2.lhe", status="replace", action="readwrite") ! only for
  open(unit=u_P2W, status="scratch", action="readwrite")
  ! convert pythia /PYJETS/ to lhef given in MSTU(163)=u1
  call pylheo
  ! read and add lhef from u_P2W
  call shower_add_lhef_to_particle_set(particle_set, u_P2W, model, os_data)
  close(u_W2P)
  close(u_P2W)
  valid = .true.
end if
end subroutine apply_PYTHIAhadronization
end subroutine apply_shower_particle_set

read in the lhe file opened in unit u and add the final particles to the particle_set

```

, the outgoing particles of the existing particle\_set are set to be virtual before the outgoing particles from the lhef are added to the particle\_set as outgoing particles. At the same time, it is searched for pairs of existing and new particles with equal momenta and PDG code, these pairs are marked as connected. All remaining existing former outgoing particles are set to be the mothers of all remaining new outgoing particles

*(Shower: procedures)*+≡

```
subroutine shower_add_lhef_to_particle_set (particle_set, u, model_in, os_data)
  type(particle_set_t), intent(inout) :: particle_set
  integer, intent(in) :: u
  type(model_t), intent(in), pointer :: model_in
  type(model_t), target, save :: model_SM_hadrons
  type(model_t), pointer :: model
  logical, save :: model_SM_hadrons_associated = .false.
  type(os_data_t), intent(in) :: os_data
  logical :: exist_SM_hadrons
  type(string_t) :: filename
  type(flavor_t) :: flv
  type(color_t) :: col
```

```
integer :: newsize, oldsize
type(particle_t), dimension(:), allocatable :: temp_prt
integer :: i, j
integer :: n_available_parents;
integer, dimension(:), allocatable :: available_parents
integer, dimension(:), allocatable :: available_children
logical, dimension(:), allocatable :: direct_child
type(vector4_t) :: diffmomentum
integer, PARAMETER :: MAXLEN=200
CHARACTER*(MAXLEN) STRING
integer ibeg
INTEGER :: NUP,IDPRUP,IDUP,ISTUP
real(kind=double) :: XWGTUP,SCALUP,AQEDUP,AQCDUP,VTIMUP,SPINUP
integer :: MOTHUP(1:2),ICOLUP(1:2)
real(kind=double) :: PUP(1:5)
real(kind=default) :: pup_dum(1:5)
character*5 buffer
```

```
CHARACTER*6 STRFMT
STRFMT='(A000)'
WRITE(STRFMT(3:5),'(I3)') MAXLEN
```

```
rewind(u)
```

```
! get newsize of particle_set, newsize = old size of particle_set + #entries - 2 (incoming par
oldsize=particle_set_get_n_tot(particle_set)
```

```
! Loop until finds line beginning with "<event>" or "<event ".
```

```
do
```

```
  READ(u,*,END=501,ERR=502) STRING
```

```
  IBEG=0
```

```
  do
```

```
    IBEG=IBEG+1
```

```
    ! Allow indentation.
```

```

        IF (STRING (IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-6) cycle
        exit
    end do
    IF (STRING (IBEG:IBEG+6).NE.'<event>'.AND.STRING (IBEG:IBEG+6).NE.'<event >') cycle
    exit
end do
! Read first line of event info -> number of entries
read (u, *, END=503, ERR=504) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
newsize = oldsize + NUP - 2
allocate (temp_prt (1:newsize))

allocate (available_parents (1:oldsize))
available_parents = 0
do i=1, particle_set_get_n_tot (particle_set)
    temp_prt (i) = particle_set_get_particle (particle_set, i)
    if (particle_get_status (temp_prt (i)) == PRT_OUTGOING .or. &
        particle_get_status (temp_prt (i)) == PRT_BEAM_REMNANT) then
        call particle_reset_status (temp_prt (i), PRT_VIRTUAL)
        available_parents (i) = i
    end if
end do

allocate (available_children (1:newsize))
allocate (direct_child (1:newsize))
available_children = 0
direct_child = .false.

! transfer particles from lhef to particle_set
!...Read NUP subsequent lines with information on each particle.
DO I = 1, NUP
    READ (u,*,END=200,ERR=505) IDUP, ISTUP, MOTHUP(1), MOTHUP(2), &
        ICOLUP(1), ICOLUP(2), (PUP (J),J=1,5), VTIMUP, SPINUP
    if ((I.eq.1).or.(I.eq.2)) cycle

    call particle_reset_status (temp_prt(oldsize+i-2), PRT_OUTGOING)
    ! particle_set%prt(oldsize+i-2)%polarization=0 ! =PRT_UNPOLARIZED !??
    if (model_test_particle (model_in, IDUP)) then
        model => model_in
    else
        ! prepare model_SM_hadrons for hadrons created in the hadronization
        ! and not present in the model file
        if (.not. model_SM_hadrons_associated) then
            call os_data_init(os_data)
            filename = "SM_hadrons.mdl"
            call model_read (model_SM_hadrons, filename, os_data, &
                exist_SM_hadrons)
            model_SM_hadrons_associated = .true.
        end if
        if (model_test_particle (model_SM_hadrons, IDUP)) then
            model => model_SM_hadrons
        else
            write (buffer, "(I5)") IDUP
            call msg_error ("Parton " // buffer // &
                " found neither in given model file nor in SM_hadrons")
        end if
    end if
end do

```

```

        return
    end if
end if
call flavor_init (flv, IDUP, model)
call particle_set_flavor (temp_prt (oldsize+i-2), flv)

if(IABS(IDUP).eq.2212 .or. IABS(IDUP).eq.2112) then
    ! PYTHIA sometimes sets color indices for protons and neutrons (?)
    ICOLUP (1) = 0
    ICOLUP (2) = 0
end if
call color_init_col_acl (col, ICOLUP (1), ICOLUP (2))
call particle_set_color (temp_prt (oldsize+i-2), col)
!particle_set%prt(oldsize+i-2)%hel=?
!particle_set%prt(oldsize+i-2)%pol=?
pup_dum = PUP
call particle_set_momentum (temp_prt (oldsize+i-2), vector4_moving (pup_dum (4), &
    vector3_moving ((/ pup_dum (1), pup_dum (2), pup_dum (3)/))))

available_children (oldsize+i-2) = oldsize+i-2
!! search for an existing particle with the same momentum -> treat these as mother and daug
do j=1, size (available_parents)
    if (available_parents (j) .eq. 0) cycle
    diffmomentum = particle_get_momentum (temp_prt (available_parents (j))) - &
        particle_get_momentum (temp_prt(oldsize+i-2))
    if(abs(diffmomentum**2) < 1D-10 .and. &
        particle_get_pdg (temp_prt (available_parents (j))).eq. &
        particle_get_pdg (temp_prt (oldsize+i-2))) then
        direct_child (available_parents (j)) = .true.
        direct_child (oldsize+i-2) = .true.
        call particle_set_parents (temp_prt (oldsize+i-2), (/ available_parents(j) /) )
        call particle_set_children (temp_prt (available_parents(j)), (/ oldsize+i-2 /) )
        available_parents (j) = 0
        available_children (oldsize+i-2) = 0
    end if
end do
end do

! remove zeros in available parents and available children
available_parents = pack (available_parents , available_parents /= 0)
available_children = pack (available_children, available_children /= 0)

do i=1, size(available_parents)
    if (direct_child (available_parents (i))) cycle
    call particle_set_children (temp_prt (available_parents (i)), available_children)
end do
do i = oldsize+1, newsize
    if (direct_child (i)) cycle
    call particle_set_parents(temp_prt(i), available_parents)
end do

! transfer to particle_set
call particle_set_replace (particle_set, temp_prt)
if (allocated (available_children)) deallocate (available_children)

```



```

        if (allocated (available_parents)) deallocate (available_parents)
        deallocate (direct_child)
        deallocate (temp_prt)
        call model_final (model_SM_hadrons)

200 continue
    return

501 write(*,*) "READING LHEF failed 501"
    return
502 write(*,*) "READING LHEF failed 502"
    return
503 write(*,*) "READING LHEF failed 503"
    return
504 write(*,*) "READING LHEF failed 504"
    return
505 write(*,*) "READING LHEF failed 504"
    return
    end subroutine shower_add_lhef_to_particle_set

<Shower: procedures>+≡
!!!!!!!!!!!!PYTHIA STYLE!!!!!!!!!!!!!!
!!! originally PYLHEF subroutine from PYTHIA 6.4.22

!C...Write out the showered event to a Les Houches Event File.
!C...Take MSTP(161) as the input for <init>...</init>

        SUBROUTINE PYLHEO

!C...Double precision and integer declarations.
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)

!C...PYTHIA commonblock: only used to provide read/write units and version.
        COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
        COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
        SAVE /PYPARS/
        SAVE /PYJETS/

!C...User process initialization commonblock.
        INTEGER MAXPUP
        PARAMETER (MAXPUP=100)
        INTEGER IDBMUP,PDFGUP,PDFSUP,IDWTUP,NPRUP,LPRUP
        DOUBLE PRECISION EBMUP,XSECUP,XERRUP,XMAXUP
        COMMON/HEPRUP/IDBMUP(2),EBMUP(2),PDFGUP(2),PDFSUP(2),IDWTUP,NPRUP,XSECUP(MAXPUP),XERRUP(MAXPUP)
        SAVE /HEPRUP/

!C...User process event common block.
        INTEGER MAXNUP
        PARAMETER (MAXNUP=500)
        INTEGER NUP,IDPRUP,IDUP,ISTUP,MOTHUP,ICOLUP
        PARAMETER (KSUSY1=1000000,KSUSY2=2000000,KTECHN=3000000, &
            KEXCIT=4000000,KDIMEN=5000000)
        DOUBLE PRECISION XWGTUP,SCALUP,AQEDUP,AQCDUP,PUP,VTIMUP,SPINUP
        COMMON/HEPEUP/NUP,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP,IDUP(MAXNUP),ISTUP(MAXNUP),MOTHUP(2,MAXNUP)

```

```

                                PUP(5,MAXNUP),VTIMUP(MAXNUP),SPINUP(MAXNUP)
SAVE /HEPEUP/

!C...Lines to read in assumed never longer than 200 characters.
PARAMETER (MAXLEN=200)
CHARACTER*(MAXLEN) STRING

INTEGER LEN

!C...Format for reading lines.
CHARACTER*6 STRFMT
STRFMT='(A000)'
WRITE(STRFMT(3:5),'(I3)') MAXLEN

!C...Rewind initialization and event files.
REWIND MSTP(161)
REWIND MSTP(162)

!C...Write header info.
WRITE(MSTP(163),'(A)') '<LesHouchesEvents version="1.0">'
WRITE(MSTP(163),'(A)') '<!--'
WRITE(MSTP(163),'(A,I1,A1,I3)') 'File generated with PYTHIA ',MSTP(181),'.',MSTP(182)
WRITE(MSTP(163),'(A)') ' and the WHIZARD2 interface'
WRITE(MSTP(163),'(A)') '-->'

!C...Loop until finds line beginning with "<init>" or "<init ".
100 READ(MSTP(161),STRFMT,END=400,ERR=400) STRING
IBEG=0
110 IBEG=IBEG+1
!C...Allow indentation.
IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-5) GOTO 110
IF(STRING(IBEG:IBEG+5).NE.'<init>'.AND.STRING(IBEG:IBEG+5).NE.'<init ') GOTO 100

!C...Read first line of initialization info and get number of processes.
READ(MSTP(161),'(A)',END=400,ERR=400) STRING
READ(STRING,*,ERR=400) IDBMUP(1),IDBMUP(2),EBMUP(1),EBMUP(2),PDFGUP(1),PDFGUP(2),PDFSUP(1),P

!C...Copy initialization lines, omitting trailing blanks.
!C...Embed in <init> ... </init> block.
WRITE(MSTP(163),'(A)') '<init>'
DO IPR=0,NPRUP
  IF(IPR.GT.0) READ(MSTP(161),'(A)',END=400,ERR=400) STRING
  LEN=MAXLEN+1
120  LEN=LEN-1
  IF(LEN.GT.1.AND.STRING(LEN:LEN).EQ.' ') GOTO 120
  WRITE(MSTP(163),'(A)',ERR=400) STRING(1:LEN)
end DO
WRITE(MSTP(163),'(A)') '</init>'

!!!! Find the numbers of entries of the <event block>
NENTRIES = 2      ! incoming partons (nearest to the beam particles)
DO I=1,N
  if((K(I,1).eq.1) .or. (K(I,1).eq.2)) then
    if(P(I,4) < 1D-10) cycle

```

```

        NENTRIES = NENTRIES + 1
    end if
end DO

!C...Begin an <event> block. Copy event lines, omitting trailing blanks.
WRITE(MSTP(163),'(A)') '<event>'
WRITE(MSTP(163),*) NENTRIES,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP

DO I=3,4      ! the incoming partons nearest to the beam particles
    WRITE(MSTP(163),*) K(I,2),-1,0,0,0,0,(P(I,J),J=1,5),0, -9
end DO
NDANGLING_COLOR = 0
NCOLOR = 0
NDANGLING_ANTIC = 0
NANTIC = 0
NNEXTC = 1    ! TODO find next free color number ??
DO I=1,N
    if((K(I,1).eq.1) .or. (K(I,1).eq.2)) then
        if(P(I,4) < 1D-10) cycle    ! workaround for zero energy photon in electron ISR
        if((K(I,2).eq.21).or.(IABS(K(I,2)).le.8).or.(IABS(K(I,2)).GE.KSUSY1+1.AND.IABS(K(I,2))
            (IABS(K(I,2)).GE.KSUSY2+1.AND.IABS(K(I,2)).LE.KSUSY2+8).or. &
            (IABS(K(I,2)).GE.1000.AND.IABS(K(I,2)).le.9999) ) then
            if(NDANGLING_COLOR.eq.0 .and. NDANGLING_ANTIC.eq.0) then
                ! new color string
                if(K(I,2).eq.21 .or. K(I,2).eq.1000021) then ! Gluon and gluino only color octet
                    NCOLOR = NNEXTC
                    NDANGLING_COLOR = NCOLOR
                    NNEXTC = NNEXTC + 1
                    NANTIC = NNEXTC
                    NDANGLING_ANTIC = NANTIC
                    NNEXTC = NNEXTC + 1
                elseif(K(I,2) .gt. 0) then ! particles to have color
                    NCOLOR = NNEXTC
                    NDANGLING_COLOR = NCOLOR
                    NANTIC = 0
                    NNEXTC = NNEXTC + 1
                elseif(K(I,2) .lt. 0) then ! antiparticles to have anticolor
                    NANTIC = NNEXTC
                    NDANGLING_ANTIC = NANTIC
                    NCOLOR = 0
                    NNEXTC = NNEXTC + 1
                end if
            else if(K(I,1).eq.1) then
                ! end of string
                NCOLOR = NDANGLING_ANTIC
                NANTIC = NDANGLING_COLOR
                NDANGLING_COLOR = 0
                NDANGLING_ANTIC = 0
            else
                ! inside the string
                if(NDANGLING_COLOR .ne. 0) then
                    NANTIC = NDANGLING_COLOR
                    NCOLOR = NNEXTC
                    NDANGLING_COLOR = NNEXTC
                end if
            end if
        end if
    end if
end DO

```

```

        NNEXTC = NNEXTC +1
    else if (NDANGLING_ANTIC .ne. 0) then
        NCOLOR = NDANGLING_ANTIC
        NANTIC = NNEXTC
        NDANGLING_ANTIC = NNEXTC
        NNEXTC = NNEXTC +1
    else
        print *, "ERROR IN PYLHEO"
    end if
end if
else
    NCOLOR = 0
    NANTIC = 0
end if

    !! As no intermediate are given out here, assume the incoming partons to be the mother
    WRITE(MSTP(163),*) K(I,2),1,1,2,NCOLOR,NANTIC,(P(I,J),J=1,5),0, -9
end if
end DO

!C..End the <event> block. Loop back to look for next event.
    WRITE(MSTP(163),'(A)') '</event>'

!C...Successfully reached end of event loop: write closing tag
!C...and remove temporary intermediate files (unless asked not to).
320  WRITE(MSTP(163),'(A)') '</LesHouchesEvents>'
    RETURN

!!C...Error exit.
400  WRITE(*,*) ' PYLHEO file joining failed!'

    RETURN
END SUBROUTINE PYLHEO

```

## 20.6 Whizard-C-Interface

```

<CCC Commands: public>≡
    public :: command_t
    public :: cmd_simulate_t
    public :: command_execute
    public :: command_final

<CCC Simulations: public>≡
    public :: checkpointing_msg_start
    public :: simulation_read_event_raw
    public :: simulation_read_event_hepmc
    public :: simulation_recover_process
    public :: simulation_recalculate
    public :: simulation_decay
    public :: simulation_select_process
    public :: simulation_handle_event
    public :: simulation_final_event

```

```

<whizard-c-interface.f90>≡
  <File header>

  <Whizard-C-Interface: Internals>
  <Whizard-C-Interface: Init and Finalize>
  <Whizard-C-Interface: Interfaced Commads>
  <Whizard-C-Interface: HepMC>

<Whizard-C-Interface: Internals>≡
  subroutine c_whizard_convert_string(c_string, f_string)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!

    implicit none

    character(kind=c_char), intent(in) :: c_string(*)
    type(string_t), intent(inout) :: f_string
    character(len=1) :: dummy_char
    integer :: dummy_i = 1

    f_string = ""
    do
      if(c_string(dummy_i) == c_null_char) then
        exit
      else if(c_string(dummy_i) == c_new_line) then
        dummy_char = CHAR(13)
        f_string = f_string // dummy_char
        dummy_char = CHAR(10)
      else
        dummy_char = c_string(dummy_i)
      end if
      f_string = f_string // dummy_char
      dummy_i = dummy_i + 1
    end do
    dummy_i = 1
  end subroutine c_whizard_convert_string

  subroutine c_whizard_commands(cmds)
    use iso_varying_string, string_t => varying_string !NODEP!
    use commands
    use diagnostics !NODEP!
    use lexers
    use models
    use parser
    use whizard

    type(string_t) :: cmds
    type(parse_tree_t) :: parse_tree
    type(parse_node_t), pointer :: pn_root
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    type(command_list_t), pointer :: cmd_list

```

```

call lexer_init_cmd_list (lexer)

call stream_init (stream, cmds)
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
pn_root => parse_tree_get_root_ptr (parse_tree)

allocate(cmd_list)
call command_list_compile(cmd_list, pn_root ,global)
call command_list_execute(cmd_list, global)
call command_list_final(cmd_list)

call parse_tree_final (parse_tree)
call stream_final (stream)
call lexer_final (lexer)
end subroutine c_whizard_commands

<Whizard-C-Interface: Init and Finalize>≡
subroutine c_whizard_init() bind(c)
  use, intrinsic :: iso_c_binding
  use diagnostics !NODEP!
  use unit_tests
  use ifiles
  use iso_varying_string, string_t => varying_string !NODEP!
  use limits, only: CMDLINE_ARG_LEN !NODEP!
  use os_interface
  use system_dependencies !NODEP!
  use whizard

  implicit none

  ! Main program variable declarations
  character(CMDLINE_ARG_LEN) :: arg
  character(2) :: option
  type(string_t) :: long_option, value
  integer :: i, j, arg_len, arg_status
  logical :: look_for_options
  logical :: interactive
  type(string_t) :: files, this, model, libname, library, libraries, logfile
  type(string_t) :: check, checks
  type(test_results_t) :: test_results
  logical :: user_code_enable = .false.
  integer :: n_user_src = 0, n_user_lib = 0
  type(string_t) :: user_src, user_lib
  type(paths_t) :: paths
  logical :: rebuild_library, rebuild_user
  logical :: rebuild_phs, rebuild_grids, rebuild_events
  logical :: recompile_library
  logical :: time_estimate
  type(ifile_t) :: commands
  type(string_t) :: command

  ! Exit status
  logical :: quit = .false.

```

```

integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
libname = "processes"
library = ""
logfile = "whizard.log"
libraries = ""
check = ""
checks = ""
user_src = ""
user_lib = ""
rebuild_library = .false.
rebuild_user = .false.
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
recompile_library = .false.
time_estimate = .true.
call paths_init (paths)

! Overall initialization
if (logfile /= "") call logfile_init (logfile)
call mask_term_signals ()
call msg_banner ()
call whizard_init &
  (preload_model=model, preload_libs=libraries, default_lib=libname, &
   rebuild_library=rebuild_library, &
   rebuild_user=rebuild_user, &
   rebuild_phs=rebuild_phs, &
   rebuild_grids=rebuild_grids, &
   rebuild_events=rebuild_events, &
   recompile_library=recompile_library, &
   time_estimate=time_estimate, &
   paths=paths, &
   user_code_enable=user_code_enable, &
   n_user_src=n_user_src, user_src=user_src, &
   n_user_lib=n_user_lib, user_lib=user_lib)

! Run any self-checks (and no commands)
if (checks /= "") then
  checks = trim (adjustl (checks))
  RUN_CHECKS: do while (checks /= "")
    call split (checks, check, " ")
    call whizard_check (check, LHAPDF_AVAILABLE, test_results)
  end do RUN_CHECKS
  call test_results%wrapup (6)
end if
end subroutine c_whizard_init

subroutine c_whizard_finalize() bind(c)

```

```

use, intrinsic :: iso_c_binding
use iso_varying_string, string_t => varying_string !NODEP!
use system_dependencies !NODEP!
use limits, only: CMDLINE_ARG_LEN !NODEP!
use diagnostics !NODEP!
use ifiles
use os_interface
use whizard

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Overall finalization
! call ifile_final (commands)
call whizard_final ()
call terminate_now_if_signal ()
call release_term_signals ()
call msg_terminate (quit_code = quit_code)
end subroutine c_whizard_finalize

subroutine c_whizard_process_string(c_cmds_in) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_cmds_in(*)
  type(string_t) :: f_cmds

  call c_whizard_convert_string(c_cmds_in, f_cmds)
  call c_whizard_commands(f_cmds)
end subroutine c_whizard_process_string

(Whizard-C-Interface: Interfaced Commands)≡
subroutine c_whizard_model(c_model) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_model(*)
  type(string_t) :: model, mdl_str

  call c_whizard_convert_string(c_model, model)
  mdl_str = "model = " // model
  call c_whizard_commands(mdl_str)
end subroutine c_whizard_model

subroutine c_whizard_library(c_library) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

```



```

character(kind=c_char) :: c_library(*)
type(string_t) :: library, lib_str

call c_whizard_convert_string(c_library, library)
lib_str = "library = " // library
call c_whizard_commands(lib_str)
end subroutine c_whizard_library

subroutine c_whizard_process(c_id, c_in, c_out) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_id(*), c_in(*), c_out(*)
  type(string_t) :: proc_str, id, in, out

  call c_whizard_convert_string(c_id, id)
  call c_whizard_convert_string(c_in, in)
  call c_whizard_convert_string(c_out, out)
  proc_str = "process " // id // " = " // in // " => " // out
  call c_whizard_commands(proc_str)
end subroutine c_whizard_process

subroutine c_whizard_compile() bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  type(string_t) :: cmp_str
  cmp_str = "compile"
  call c_whizard_commands(cmp_str)
end subroutine c_whizard_compile

subroutine c_whizard_load(c_library) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_library(*)
  type(string_t) :: library, load_str

  call c_whizard_convert_string(c_library, library)
  load_str = "load = " // library
  call c_whizard_commands(load_str)
end subroutine c_whizard_load

subroutine c_whizard_beams(c_specs) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

```

```

character(kind=c_char) :: c_specs(*)
type(string_t) :: specs, beam_str

call c_whizard_convert_string(c_specs, specs)
beam_str = "beams = " // specs
call c_whizard_commands(beam_str)
end subroutine c_whizard_beams

subroutine c_whizard_integrate(c_process) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, int_str

  call c_whizard_convert_string(c_process, process)
  int_str = "integrate (" // process // ")"
  call c_whizard_commands(int_str)
end subroutine c_whizard_integrate

subroutine c_whizard_matrix_element_test(c_process) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, me_str

  call c_whizard_convert_string(c_process, process)
  me_str = "matrix_element_test (" // process // ")"
  call c_whizard_commands(me_str)
end subroutine c_whizard_matrix_element_test

subroutine c_whizard_simulate(c_id) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  character(kind=c_char) :: c_id(*)
  type(string_t) :: sim_str, id

  call c_whizard_convert_string(c_id, id)
  sim_str = "simulate (" // id // ")"
  call c_whizard_commands(sim_str)
end subroutine c_whizard_simulate

subroutine c_whizard_sqrts(c_value, c_unit) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

```

```

implicit none

character(kind=c_char) :: c_unit(*)
integer(kind=c_int) :: c_value
integer :: f_value
character(len=8) :: f_val
type(string_t) :: val, unit, sqrts_str

f_value = c_value
write(f_val,'(i8)') f_value
val = f_val
call c_whizard_convert_string(c_unit, unit)
sqrts_str = "sqrts =" // val // unit
call c_whizard_commands(sqrts_str)
end subroutine c_whizard_sqrts

{Whizard-C-Interface: HepMC}≡
type(c_ptr) function c_whizard_hepmc_test(c_id, c_proc_id, c_event_id) bind(c)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!
  use commands
  use diagnostics !NODEP!
  use events
  use hepmc_interface
  use lexers
  use limits, only: MAX_TRIES_FOR_SINGLE_EVENT !NODEP!
  use models
  use parser
  use processes
  use rt_data
  use simulations
  use whizard
  use os_interface

  implicit none

  type(string_t) :: sim_str
  type(parse_tree_t) :: parse_tree
  type(parse_node_t), pointer :: pn_root
  type(stream_t), target :: stream
  type(lexer_t) :: lexer
  type(command_list_t), pointer :: cmd_list
  type(command_t), pointer :: command
  type(os_data_t) :: os_data

  type(cmd_simulate_t), target :: simulate
  logical :: ok !, mlm_matching
  integer :: i_evt
  type(simulation_t), target :: sim

  character(kind=c_char), intent(in) :: c_id(*)
  type(string_t) :: id
  integer(kind=c_int), value :: c_proc_id, c_event_id
  integer :: proc_id, event_id

```

```

type(hepmc_event_t), pointer :: hepmc_event

type(process_t), pointer :: process
integer :: proc

integer :: factorization_mode, try

call c_whizard_convert_string(c_id, id)
sim_str = "simulate (" // id // ")"

proc_id = c_proc_id
event_id = c_event_id

allocate(hepmc_event)
call hepmc_event_init (hepmc_event, c_proc_id, c_event_id)

call lexer_init_cmd_list (lexer)

call stream_init (stream, sim_str)
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
pn_root => parse_tree_get_root_ptr (parse_tree)

allocate(cmd_list)
call command_list_compile(cmd_list, pn_root ,global)

command => cmd_list%first
simulate = command%simulate

call rt_data_link (simulate%local, global)

if (associated (simulate%options)) then
    call command_list_execute (simulate%options, simulate%local)
end if

call simulation_init (sim, simulate%process_id, simulate%local, global%var_list, ok, verbose=.fa

if (ok) then
    call simulation_setup_reweight &
        (sim, simulate%local%pn_reweight_expr, verbose=.false.)
    call simulation_select_process (sim, simulate%local%rng, process, proc)

    if (sim%allow_decays) then
        call event_init (sim%event, process, &
            sim%event_vars, sim%decay_tree(proc))
    else
        call event_init (sim%event, process, sim%event_vars)
    end if

    if (sim%use_num_id) then
        sim%event_vars%process_num_id = sim%num_id(proc)
    else
        sim%event_vars%process_num_id = proc
    end if
end if

```

```

end if

if (sim%spar%polarized) then
    factorization_mode = FM_SELECT_HELICITY
else
    factorization_mode = FM_IGNORE_HELICITY
end if

call os_data_init(os_data)
GENERATE: do try = 1, MAX_TRIES_FOR_SINGLE_EVENT
    call event_generate &
        (sim%event, simulate%local%rng, sim%spar%unweighted, &
        factorization_mode, &
        keep_correlations=.false., &
        keep_virtual=.true., os_data=os_data, &
        shower_settings = sim%spar%shower_settings)
    if(event_is_vetoed(sim%event).and. &
        (.not.sim%n_events_set)) then
        sim%n_events = sim%n_events - 1
        if(sim%i_evt .ge. sim%n_events) then
            call event_final(sim%event)
            return
        end if
    end if
    if (event_is_valid (sim%event).and. &
        (.not.event_is_vetoed(sim%event))) exit GENERATE
end do GENERATE
if (.not. event_is_valid (sim%event)) then
    write (msg_buffer, "(A,I0,A)") "Failed to generate a valid event " &
        // "after ", MAX_TRIES_FOR_SINGLE_EVENT, " tries"
    call msg_fatal ( )
end if

call event_renormalize_weight (sim%event, sim%norm_weight)

call event_write_to_hepmc(sim%event, hepmc_event)

call simulation_handle_event (sim)
call simulation_final_event (sim)
call simulation_final (sim, verbose=.false.)
end if
call rt_data_restore (global, simulate%local)

call command_list_final(cmd_list)

call parse_tree_final (parse_tree)
call stream_final (stream)
call lexer_final (lexer)

c_whizard_hepmc_test = c_loc(hepmc_event)
return
end function c_whizard_hepmc_test

```

## Chapter 21

# Cross References

21.1 Identifiers

21.2 Chunks