Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 2.0

ESMF Joint Specification Team: V. Balaji, Byron Boville, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Brian Eaton, Bob Hallberg, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Brian Kauffman, Jay Larson, John Michalakes, David Neckels, Chuck Panaccione, Jim Rosinski, Earl Schwab, Shepard Smithline, Max Suarez, Spencer Swift, Gerhard Theurich, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky

26th January 2005

NASA Earth Science Technology Office
Computational Technologies Project
CAN 00-OES-01
http://www.esmf.ucar.edu

Acknowledgements

The ESMF software is based on the contributions of a broad community. Below are the software packages that are included in ESMF or strongly influenced our design. We'd like to express our gratitude to the developers of these codes for access to their software as well as their ideas and advice.

- The Spherical Coordinate Remapping and Interpolation Package (SCRIP) from Los Alamos, on which we based our regridding functionality with the help of SCRIP author Phil Jones
- The Inpack configuration attributes package from NASA Goddard, which was adapted for use in ESMF by members of NASA Global Modeling and Assimilation group
- The Flexible Modeling System (FMS) package from GFDL and the Goddard Earth Modeling System (GEMS) from NASA Goddard, both of which provided inspiration for overall ESMF structure
- The Weather Research and Forecast (WRF) modeling system, on which we based our underlying I/O implementation
- The Common Component Architecture (CCA) effort within the DoE, from which we drew many ideas about how to design components
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system

Contents

I ESMF Overview **18**

1 Release Notes **19**

2 What is the Earth System Modeling Framework? **19**

3 The ESMF Reference Manual for Fortran **19**

4 How to Contact User Support and Find Additional Information **19**

5 How to Submit New Requirements **20**

6 Conventions **20**
6.1 Document Conventions . . . 20
6.2 Method Name and Argument Conventions . . . 21
6.3 Locating Methods in this Manual . . . 21

7 The ESMF Application Programming Interface **22**
7.1 Standard Methods and Interface Rules . . . 22
7.2 Deep and Shallow Classes . . . 23
7.3 Special Methods . . . 23
7.4 The ESMF Data Hierarchy . . . 23
7.5 ESMF Spatial Classes . . . 24
7.6 ESMF DataMap Classes . . . 24
7.7 ESMF Specification Classes . . . 25
7.8 ESMF Utility Classes . . . 25

8 Overall Rules and Behavior **25**
8.1 Allocation Rules . . . 25
8.2 Attributes . . . 25

9 Integrating ESMF into Applications **26**
9.1 Using the ESMF Superstructure . . . 26
9.2 Using the ESMF Infrastructure . . . 26

10 Global Options and Parameters **27**
10.1 Flags . . . 27
10.1.1 ESMF_AllocFlag . . . 27
10.1.2 ESMF_BlockingFlag . . . 27
10.1.3 ESMF_CopyFlag . . . 27
10.1.4 ESMF_IndexFlag . . . 27
10.1.5 ESMF_InterleaveFlag . . . 27
10.1.6 ESMF_NeededFlag . . . 28
10.1.7 ESMF_ReadyFlag . . . 28
10.1.8 ESMF_ReduceFlag . . . 28
10.1.9 ESMF_ReqForRestartFlag . . . 28
10.1.10 ESMF_ValidFlag . . . 28
10.2 Parameters . . . 29
10.2.1 ESMF_DataKind . . . 29
10.2.2 ESMF_DataType . . . 29

11 Overall Design and Implementation Notes **29**

II Superstructure 30

12 Overview of Superstructure **31**
12.1 Superstructure Classes . . . 31
12.2 Distribution and Scoping of Components . . . 32
12.3 Performance . . . 33
12.4 Object Model . . . 35

13 Application Driver and Required ESMF Methods **35**
13.1 Description . . . 35
13.2 Use and Examples . . . 36
13.3 Restrictions and Future Work . . . 40
13.4 Required ESMF Methods . . . 40
13.4.1 ESMF_Initialize . . . 40
13.4.2 ESMF_Finalize . . . 41
13.4.3 User-Code SetServices Method . . . 41
13.4.4 User-Code Initialize, Run, and Finalize Methods . . . 41

14 GridComp Class **42**
14.1 Description . . . 42
14.2 GridComp Options . . . 42
14.2.1 ESMF_GridCompType . . . 42
14.3 Use and Examples . . . 43
14.3.1 Specifying a User-Code SetServices Routine . . . 43
14.3.2 Specifying a User-Code Initialize Routine . . . 44
14.3.3 Specifying a User-Code Run Routine . . . 44
14.3.4 Specifying a User-Code Finalize Routine . . . 45
14.4 Restrictions and Future Work . . . 45
14.5 Class API: Basic GridComp Methods . . . 46
14.5.1 ESMF_GridCompCreate . . . 46
14.5.2 ESMF_GridCompCreate . . . 47
14.5.3 ESMF_GridCompDestroy . . . 48
14.5.4 ESMF_GridCompFinalize . . . 48
14.5.5 ESMF_GridCompGet . . . 49
14.5.6 ESMF_GridCompInitialize . . . 50
14.5.7 ESMF_GridCompPrint . . . 50
14.5.8 ESMF_GridCompReadRestart . . . 51
14.5.9 ESMF_GridCompRun . . . 52
14.5.10 ESMF_GridCompSet . . . 52
14.5.11 ESMF_GridCompValidate . . . 53
14.5.12 ESMF_GridCompWriteRestart . . . 54
14.5.13 ESMF_GridCompWait . . . 54
14.6 Class API: SetServices and Related Methods . . . 55
14.6.1 ESMF_GridCompGetInternalState . . . 55
14.6.2 ESMF_GridCompSetEntryPoint . . . 55
14.6.3 ESMF_GridCompSetInternalState . . . 56
14.6.4 ESMF_GridCompSetServices . . . 56

15 CplComp Class **57**
15.1 Description . . . 57
15.2 Use and Examples . . . 58
15.2.1 Specifying a User-Code SetServices Routine . . . 58
15.2.2 Specifying a User-Code Initialize Routine . . . 59
15.2.3 Specifying a User-Code Run Routine . . . 59
15.2.4 Specifying a User-Code Finalize Routine . . . 60
15.3 Restrictions and Future Work . . . 60
15.4 Class API: Basic CplComp Methods . . . 61
15.4.1 ESMF_CplCompCreate . . . 61
15.4.2 ESMF_CplCompCreate . . . 61
15.4.3 ESMF_CplCompDestroy . . . 62
15.4.4 ESMF_CplCompFinalize . . . 63
15.4.5 ESMF_CplCompGet . . . 63
15.4.6 ESMF_CplCompInitialize . . . 64
15.4.7 ESMF_CplCompPrint . . . 65
15.4.8 ESMF_CplCompReadRestart . . . 65
15.4.9 ESMF_CplCompRun . . . 66
15.4.10 ESMF_CplCompSet . . . 67
15.4.11 ESMF_CplCompValidate . . . 67
15.4.12 ESMF_CplCompWriteRestart . . . 68
15.4.13 ESMF_CplCompWait . . . 68
15.5 Class API: SetServices and Related Methods . . . 69
15.5.1 ESMF_CplCompGetInternalState . . . 69
15.5.2 ESMF_CplCompSetEntryPoint . . . 69
15.5.3 ESMF_CplCompSetInternalState . . . 70
15.5.4 ESMF_CplCompSetServices . . . 70

16 State Class **71**
16.1 Description . . . 71
16.2 State Options . . . 71
16.2.1 ESMF_StateItemType . . . 71
16.2.2 ESMF_StateType . . . 71
16.3 Use and Examples . . . 72
16.3.1 Empty State Create . . . 73
16.3.2 Adding Items to a State . . . 73
16.3.3 Adding Placeholders to a State . . . 73
16.3.4 Marking an Item Needed . . . 74
16.3.5 Creating a Needed Item . . . 74
16.4 Restrictions and Future Work . . . 74
16.5 Design and Implementation Notes . . . 74
16.6 Object Model . . . 75
16.7 Class API: Basic State Methods . . . 75
16.7.1 ESMF_StateAddArray . . . 75
16.7.2 ESMF_StateAddArray . . . 75
16.7.3 ESMF_StateAddBundle . . . 76
16.7.4 ESMF_StateAddBundle . . . 77
16.7.5 ESMF_StateAddField . . . 77
16.7.6 ESMF_StateAddField . . . 78
16.7.7 ESMF_StateAddNameOnly . . . 78
16.7.8 ESMF_StateAddNameOnly . . . 79

16.7.9 ESMF_StateAddState . . . 79
16.7.10 ESMF_StateAddState . . . 80
16.7.11 ESMF_StateCreate . . . 80
16.7.12 ESMF_StateDestroy . . . 81
16.7.13 ESMF_StateGet . . . 82
16.7.14 ESMF_StateGetArray . . . 82
16.7.15 ESMF_StateGetAttribute . . . 83
16.7.16 ESMF_StateGetAttribute . . . 83
16.7.17 ESMF_StateGetAttribute . . . 84
16.7.18 ESMF_StateGetAttribute . . . 85
16.7.19 ESMF_StateGetAttribute . . . 85
16.7.20 ESMF_StateGetAttribute . . . 86
16.7.21 ESMF_StateGetAttribute . . . 86
16.7.22 ESMF_StateGetAttribute . . . 87
16.7.23 ESMF_StateGetAttribute . . . 87
16.7.24 ESMF_StateGetAttribute . . . 88
16.7.25 ESMF_StateGetAttribute . . . 88
16.7.26 ESMF_StateGetAttributeCount . . . 89
16.7.27 ESMF_StateGetAttributeInfo . . . 89
16.7.28 ESMF_StateGetAttributeInfo . . . 90
16.7.29 ESMF_StateGetBundle . . . 90
16.7.30 ESMF_StateGetField . . . 91
16.7.31 ESMF_StateGetNeeded . . . 92
16.7.32 ESMF_StateGetState . . . 92
16.7.33 ESMF_StateIsNeeded . . . 93
16.7.34 ESMF_StatePrint . . . 93
16.7.35 ESMF_StateSetAttribute . . . 94
16.7.36 ESMF_StateSetAttribute . . . 94
16.7.37 ESMF_StateSetAttribute . . . 95
16.7.38 ESMF_StateSetAttribute . . . 95
16.7.39 ESMF_StateSetAttribute . . . 96
16.7.40 ESMF_StateSetAttribute . . . 96
16.7.41 ESMF_StateSetAttribute . . . 97
16.7.42 ESMF_StateSetAttribute . . . 97
16.7.43 ESMF_StateSetAttribute . . . 98
16.7.44 ESMF_StateSetAttribute . . . 98
16.7.45 ESMF_StateSetAttribute . . . 99
16.7.46 ESMF_StateSetNeeded . . . 99
16.7.47 ESMF_StateValidate . . . 100
16.8 Class API: State Overloads for Fortran Arrays . . . 100
16.8.1 ESMF_StateGetDataPointer . . . 100

III Infrastructure: Fields and Grids 102

17 Overview of Infrastructure Data Handling 103
17.1 Infrastructure Data Classes . . . 103
17.2 Object Model . . . 104
17.3 Design and Implementation Notes . . . 104

18 Bundle Class **105**
18.1 Description 105
18.2 Bundle Options 105
18.2.1 ESMF_PackFlag 105
18.3 Use and Examples 105
18.3.1 Bundle Creation 105
18.3.2 Accessing Bundle Data 105
18.3.3 Bundle Deletion 106
18.4 Restrictions and Future Work 108
18.5 Design and Implementation Notes 108
18.6 Class API: Basic Bundle Methods 109
18.6.1 ESMF_BundleAddField 109
18.6.2 ESMF_BundleAddField 109
18.6.3 ESMF_BundleCreate 110
18.6.4 ESMF_BundleCreate 110
18.6.5 ESMF_BundleDestroy 111
18.6.6 ESMF_BundleGet 111
18.6.7 ESMF_BundleGetAttribute 112
18.6.8 ESMF_BundleGetAttribute 113
18.6.9 ESMF_BundleGetAttribute 113
18.6.10 ESMF_BundleGetAttribute 114
18.6.11 ESMF_BundleGetAttribute 114
18.6.12 ESMF_BundleGetAttribute 115
18.6.13 ESMF_BundleGetAttribute 115
18.6.14 ESMF_BundleGetAttribute 116
18.6.15 ESMF_BundleGetAttribute 116
18.6.16 ESMF_BundleGetAttribute 117
18.6.17 ESMF_BundleGetAttribute 117
18.6.18 ESMF_BundleGetAttributeCount 118
18.6.19 ESMF_BundleGetAttributeInfo 118
18.6.20 ESMF_BundleGetAttributeInfo 119
18.6.21 ESMF_BundleGetField 119
18.6.22 ESMF_BundleGetField 120
18.6.23 ESMF_BundlePrint 120
18.6.24 ESMF_BundleSetAttribute 121
18.6.25 ESMF_BundleSetAttribute 121
18.6.26 ESMF_BundleSetAttribute 122
18.6.27 ESMF_BundleSetAttribute 122
18.6.28 ESMF_BundleSetAttribute 123
18.6.29 ESMF_BundleSetAttribute 123
18.6.30 ESMF_BundleSetAttribute 124
18.6.31 ESMF_BundleSetAttribute 125
18.6.32 ESMF_BundleSetAttribute 125
18.6.33 ESMF_BundleSetAttribute 126
18.6.34 ESMF_BundleSetGrid 126
18.6.35 ESMF_BundleValidate 127
18.7 Class API: Bundle Overloads for Fortran Arrays 127
18.7.1 ESMF_BundleGetDataPointer 127

19 BundleDataMap Class **128**
19.1 Description . . . 128
19.2 BundleDataMap Options . . . 128
19.3 Use and Examples . . . 128
19.3.1 Setting BundleDataMap Defaults . . . 128
19.3.2 Setting BundleDataMap Values . . . 129
19.3.3 Getting BundleDataMap Values . . . 129
19.4 Restrictions and Future Work . . . 129
19.5 Class API . . . 129
19.5.1 ESMF_BundleDataMapGet . . . 129
19.5.2 ESMF_BundleDataMapPrint . . . 130
19.5.3 ESMF_BundleDataMapSet . . . 130
19.5.4 ESMF_BundleDataMapSetDefault . . . 131
19.5.5 ESMF_BundleDataMapSetInvalid . . . 131
19.5.6 ESMF_BundleDataMapValidate . . . 131

20 Field Class **132**
20.1 Description . . . 132
20.2 Use and Examples . . . 132
20.2.1 Field Creation . . . 132
20.2.2 Field Deletion . . . 133
20.2.3 Field Create with Grid and Array . . . 133
20.2.4 Field Create with Grid and ArraySpec . . . 133
20.2.5 Empty Field Create . . . 134
20.2.6 Destroy a Field . . . 134
20.3 Restrictions and Future Work . . . 134
20.4 Design and Implementation Notes . . . 134
20.5 Class API: Basic Field Methods . . . 134
20.5.1 ESMF_FieldCreateNoData . . . 134
20.5.2 ESMF_FieldCreateNoData . . . 135
20.5.3 ESMF_FieldCreateNoData . . . 136
20.5.4 ESMF_FieldDestroy . . . 137
20.5.5 ESMF_FieldGet . . . 137
20.5.6 ESMF_FieldGetArray . . . 138
20.5.7 ESMF_FieldGetAttribute . . . 138
20.5.8 ESMF_FieldGetAttribute . . . 139
20.5.9 ESMF_FieldGetAttribute . . . 139
20.5.10 ESMF_FieldGetAttribute . . . 140
20.5.11 ESMF_FieldGetAttribute . . . 140
20.5.12 ESMF_FieldGetAttribute . . . 141
20.5.13 ESMF_FieldGetAttribute . . . 141
20.5.14 ESMF_FieldGetAttribute . . . 142
20.5.15 ESMF_FieldGetAttribute . . . 142
20.5.16 ESMF_FieldGetAttribute . . . 143
20.5.17 ESMF_FieldGetAttribute . . . 143
20.5.18 ESMF_FieldGetAttributeCount . . . 144
20.5.19 ESMF_FieldGetAttributeInfo . . . 144
20.5.20 ESMF_FieldGetAttributeInfo . . . 145
20.5.21 ESMF_FieldPrint . . . 145
20.5.22 ESMF_FieldSetArray . . . 146
20.5.23 ESMF_FieldSetAttribute . . . 146

20.5.24 ESMF_FieldSetAttribute 147
20.5.25 ESMF_FieldSetAttribute 147
20.5.26 ESMF_FieldSetAttribute 148
20.5.27 ESMF_FieldSetAttribute 148
20.5.28 ESMF_FieldSetAttribute 149
20.5.29 ESMF_FieldSetAttribute 149
20.5.30 ESMF_FieldSetAttribute 150
20.5.31 ESMF_FieldSetAttribute 150
20.5.32 ESMF_FieldSetAttribute 151
20.5.33 ESMF_FieldSetAttribute 151
20.5.34 ESMF_FieldSetGrid 152
20.5.35 ESMF_FieldSetDataMap 152
20.5.36 ESMF_FieldValidate 152
20.5.37 ESMF_FieldWrite 153
20.6 Class API: Field Overloads for Fortran Arrays 153
20.6.1 ESMF_FieldCreate 153
20.6.2 ESMF_FieldCreate 154
20.6.3 ESMF_FieldCreate 155
20.6.4 ESMF_FieldCreate 156
20.6.5 ESMF_FieldCreate 157
20.6.6 ESMF_FieldGetDataPointer 158
20.6.7 ESMF_FieldSetDataPointer 159
20.7 Class API: Field Communications 159
20.7.1 ESMF_FieldAllGather 159
20.7.2 ESMF_FieldGather 160
20.7.3 ESMF_FieldHalo 161
20.7.4 ESMF_FieldHaloRelease 161
20.7.5 ESMF_FieldHaloStore 162
20.7.6 ESMF_FieldRedist 162
20.7.7 ESMF_FieldRedistRelease 163
20.7.8 ESMF_FieldRedistStore 164
20.7.9 ESMF_FieldRedistStore 164
20.7.10 ESMF_FieldRegrid 165
20.7.11 ESMF_FieldRegridRelease 166
20.7.12 ESMF_FieldRegridStore 166
20.7.13 ESMF_FieldScatter 167

21 FieldDataMap Class **168**
21.1 Description 168
21.2 Use and Examples 168
21.2.1 Setting Field DataMap Defaults and Invalidation 169
21.2.2 Setting Field DataMap Values 169
21.2.3 Getting Field DataMap Values 169
21.3 Restrictions and Future Work 170
21.4 Design and Implementation Notes 170
21.5 Class API 170
21.5.1 ESMF_FieldDataMapGet 170
21.5.2 ESMF_FieldDataMapPrint 171
21.5.3 ESMF_FieldDataMapSet 171
21.5.4 ESMF_FieldDataMapSetDefault 172
21.5.5 ESMF_FieldDataMapSetDefault 173

21.5.6 ESMF_FieldDataMapSetInvalid 173
21.5.7 ESMF_FieldDataMapValidate 174

22 Array Class **174**
22.1 Description 174
22.2 Use and Examples 174
22.2.1 Create an Array with Existing Data 175
22.2.2 Destroy an Array 175
22.2.3 Create an Array and Copy Existing Data 175
22.2.4 Create an Array and Allocate Data Space 176
22.2.5 Print Array Contents 176
22.2.6 Get a Pointer to the Array Contents 177
22.2.7 Destroy an Array 177
22.2.8 Get a Pointer to a Copy of the Array Contents 177
22.3 Restrictions and Future Work 177
22.4 Design and Implementation Notes 177
22.5 Class API: Basic Array Methods 178
22.5.1 ESMF_ArrayGet 178
22.5.2 ESMF_ArrayGetAttribute 179
22.5.3 ESMF_ArrayGetAttribute 180
22.5.4 ESMF_ArrayGetAttribute 180
22.5.5 ESMF_ArrayGetAttribute 181
22.5.6 ESMF_ArrayGetAttribute 182
22.5.7 ESMF_ArrayGetAttribute 182
22.5.8 ESMF_ArrayGetAttribute 183
22.5.9 ESMF_ArrayGetAttribute 183
22.5.10 ESMF_ArrayGetAttribute 184
22.5.11 ESMF_ArrayGetAttribute 184
22.5.12 ESMF_ArrayGetAttribute 185
22.5.13 ESMF_ArrayGetAttributeCount 185
22.5.14 ESMF_ArrayGetAttributeInfo 186
22.5.15 ESMF_ArrayGetAttributeInfo 186
22.5.16 ESMF_ArrayPrint 187
22.5.17 ESMF_ArraySetAttribute 187
22.5.18 ESMF_ArraySetAttribute 188
22.5.19 ESMF_ArraySetAttribute 189
22.5.20 ESMF_ArraySetAttribute 189
22.5.21 ESMF_ArraySetAttribute 190
22.5.22 ESMF_ArraySetAttribute 190
22.5.23 ESMF_ArraySetAttribute 191
22.5.24 ESMF_ArraySetAttribute 191
22.5.25 ESMF_ArraySetAttribute 192
22.5.26 ESMF_ArraySetAttribute 192
22.5.27 ESMF_ArraySetAttribute 193
22.5.28 ESMF_ArraySet 193
22.5.29 ESMF_ArrayValidate 194
22.5.30 ESMF_ArrayWrite 194
22.6 Class API: Array Overloads for Fortran Arrays 195
22.6.1 ESMF_ArrayCreate 195
22.6.2 ESMF_ArrayCreate 195
22.6.3 ESMF_ArrayCreate 196

22.6.4 ESMF_ArrayCreate . . . 197
22.6.5 ESMF_ArrayCreate . . . 198
22.6.6 ESMF_ArrayGetData . . . 199
22.7 Class API: Array Communications . . . 199
22.7.1 ESMF_ArrayGather . . . 199
22.7.2 ESMF_ArrayHalo . . . 200
22.7.3 ESMF_ArrayHaloRelease . . . 200
22.7.4 ESMF_ArrayHaloStore . . . 201
22.7.5 ESMF_ArrayRedist . . . 201
22.7.6 ESMF_ArrayRedistRelease . . . 202
22.7.7 ESMF_ArrayRedistStore . . . 202

23 ArrayDataMap Class 203
23.1 Description . . . 203
23.2 ArrayDataMap Options . . . 203
23.2.1 ESMF_IndexOrder . . . 203
23.2.2 ESMF_RelLoc . . . 204
23.3 Use and Examples . . . 205
23.3.1 Setting Array DataMap Defaults and Invalidation . . . 205
23.3.2 Setting Array DataMap Values . . . 206
23.3.3 Getting Array DataMap Values . . . 206
23.4 Restrictions and Future Work . . . 206
23.5 Class API . . . 206
23.5.1 ESMF_ArrayDataMapGet . . . 206
23.5.2 ESMF_ArrayDataMapPrint . . . 207
23.5.3 ESMF_ArrayDataMapSet . . . 207
23.5.4 ESMF_ArrayDataMapSetDefault . . . 208
23.5.5 ESMF_ArrayDataMapSetDefault . . . 209
23.5.6 ESMF_ArrayDataMapSetInvalid . . . 209
23.5.7 ESMF_ArrayDataMapValidate . . . 210

24 ArraySpec Class 210
24.1 Description . . . 210
24.2 Use and Examples . . . 210
24.2.1 Setting ArraySpec Values . . . 211
24.2.2 Getting ArraySpec Values . . . 211
24.3 Restrictions and Future Work . . . 211
24.4 Design and Implementation Notes . . . 212
24.5 Class API . . . 212
24.5.1 ESMF_ArraySpecGet . . . 212
24.5.2 ESMF_ArraySpecSet . . . 212

25 Grid Class 213
25.1 Description . . . 213
25.2 Grid Options . . . 213
25.2.1 ESMF_CoordIndex . . . 213
25.2.2 ESMF_CoordOrder . . . 213
25.2.3 ESMF_CoordSystem . . . 214
25.2.4 ESMF_GridHorzStagger . . . 214
25.2.5 ESMF_GridType . . . 215
25.2.6 ESMF_GridVertStagger . . . 216

25.2.7 ESMF_GridVertType . . . 216
25.3 Use and Examples . . . 216
25.3.1 Uniform 2D Grid Creation . . . 217
25.3.2 3D Grid Creation . . . 218
25.4 Restrictions and Future Work . . . 218
25.5 Design and Implementation Notes . . . 220
25.5.1 Grid Classes . . . 220
25.5.2 DistGrid Implementation Notes . . . 220
25.5.3 PhysGrid Implementation Notes . . . 221
25.6 Object Model . . . 221
25.7 Class API: General Grid Methods . . . 222
25.7.1 ESMF_GridAddVertHeight . . . 222
25.7.2 ESMF_GridCreate . . . 223
25.7.3 ESMF_GridDestroy . . . 224
25.7.4 ESMF_GridDistribute . . . 224
25.7.5 ESMF_GridGet . . . 225
25.7.6 ESMF_GridGetAttribute . . . 226
25.7.7 ESMF_GridGetAttribute . . . 227
25.7.8 ESMF_GridGetAttribute . . . 227
25.7.9 ESMF_GridGetAttribute . . . 228
25.7.10 ESMF_GridGetAttribute . . . 228
25.7.11 ESMF_GridGetAttribute . . . 229
25.7.12 ESMF_GridGetAttribute . . . 229
25.7.13 ESMF_GridGetAttribute . . . 230
25.7.14 ESMF_GridGetAttribute . . . 230
25.7.15 ESMF_GridGetAttribute . . . 231
25.7.16 ESMF_GridGetAttribute . . . 231
25.7.17 ESMF_GridGetAttributeCount . . . 232
25.7.18 ESMF_GridGetAttributeInfo . . . 232
25.7.19 ESMF_GridGetAttributeInfo . . . 233
25.7.20 ESMF_GridGetCoord . . . 233
25.7.21 ESMF_GridGetDELocalInfo . . . 234
25.7.22 ESMF_GridGlobalToDELocalIndex . . . 235
25.7.23 ESMF_GridDELocalToGlobalIndex . . . 236
25.7.24 ESMF_GridPrint . . . 237
25.7.25 ESMF_GridSet . . . 237
25.7.26 ESMF_GridSetAttribute . . . 238
25.7.27 ESMF_GridSetAttribute . . . 239
25.7.28 ESMF_GridSetAttribute . . . 239
25.7.29 ESMF_GridSetAttribute . . . 240
25.7.30 ESMF_GridSetAttribute . . . 240
25.7.31 ESMF_GridSetAttribute . . . 241
25.7.32 ESMF_GridSetAttribute . . . 241
25.7.33 ESMF_GridSetAttribute . . . 242
25.7.34 ESMF_GridSetAttribute . . . 242
25.7.35 ESMF_GridSetAttribute . . . 243
25.7.36 ESMF_GridSetAttribute . . . 243
25.7.37 ESMF_GridValidate . . . 244
25.8 Class API: Logically Rectangular Grid Methods . . . 244
25.8.1 ESMF_GridCreateHorzLatLon . . . 244
25.8.2 ESMF_GridCreateHorzLatLonUni . . . 245

25.8.3 ESMF_GridCreateHorzXY . . . 246
25.8.4 ESMF_GridCreateHorzXYUni . . . 248

26 IOSpec Class **249**
26.1 Description . . . 249
26.2 Use and Examples . . . 249
26.3 Restrictions and Future Work . . . 249
26.4 Class API . . . 249
26.4.1 ESMF_IOSpecGet . . . 249
26.4.2 ESMF_IOSpecSet . . . 250

27 Overview of Distributed Data Methods **251**
27.1 Higher Level Functions . . . 251
27.2 Lower Level Functions . . . 251

28 Halo Method **252**
28.1 Description . . . 252
28.2 Halo Domains . . . 252

29 Regrid Method **252**
29.1 Description . . . 252
29.2 Regrid Options . . . 252
29.2.1 ESMF_RegridMethod . . . 252
29.2.2 ESMF_RegridNormOpt . . . 260
29.3 Design and Implementation Notes . . . 260
29.3.1 Parallel Implementation . . . 261
29.3.2 Regrid Objects . . . 261
29.3.3 Restrictions and Future Work . . . 261
29.3.4 Precomputing and Executing a Regrid . . . 262

30 Redist Method **262**
30.1 Description . . . 262

IV Infrastructure: Utilities **263**

31 Overview of Infrastructure Utility Classes **264**

32 Time Manager Utility **265**
32.1 Time Manager Classes . . . 265
32.2 Calendar . . . 266
32.3 Time Instants and Time Intervals . . . 266
32.4 Clocks and Alarms . . . 266
32.5 Design and Implementation Notes . . . 267
32.6 Object Model . . . 269

33 Calendar Class **270**
33.1 Description . . . 270
33.2 Calendar Options . . . 270
33.2.1 ESMF_CalendarType . . . 270
33.3 Use and Examples . . . 270
33.3.1 Calendar Creation . . . 271

33.3.2 Calendar Comparison . . . 271
33.3.3 Time Conversion Between Calendars . . . 271
33.3.4 Calendar Destruction . . . 272
33.4 Restrictions and Future Work . . . 272
33.5 Class API . . . 272
33.5.1 ESMF_CalendarOperator(==) . . . 272
33.5.2 ESMF_CalendarOperator(==) . . . 273
33.5.3 ESMF_CalendarOperator(==) . . . 273
33.5.4 ESMF_CalendarOperator(==) . . . 274
33.5.5 ESMF_CalendarOperator(/=) . . . 274
33.5.6 ESMF_CalendarOperator(/=) . . . 275
33.5.7 ESMF_CalendarOperator(/=) . . . 275
33.5.8 ESMF_CalendarOperator(/=) . . . 276
33.5.9 ESMF_CalendarCreate . . . 276
33.5.10 ESMF_CalendarCreate . . . 277
33.5.11 ESMF_CalendarCreate . . . 277
33.5.12 ESMF_CalendarDestroy . . . 278
33.5.13 ESMF_CalendarGet . . . 278
33.5.14 ESMF_CalendarPrint . . . 279
33.5.15 ESMF_CalendarSet . . . 280
33.5.16 ESMF_CalendarSet . . . 281
33.5.17 ESMF_CalendarSetDefault . . . 281
33.5.18 ESMF_CalendarSetDefault . . . 282
33.5.19 ESMF_CalendarValidate . . . 282

34 Time Class 284
34.1 Description . . . 284
34.2 Use and Examples . . . 284
34.2.1 Time Initialization . . . 284
34.2.2 Time Increment . . . 285
34.2.3 Time Comparison . . . 285
34.3 Restrictions and Future Work . . . 285
34.4 Class API . . . 286
34.4.1 ESMF_TimeOperator(+) . . . 286
34.4.2 ESMF_TimeOperator(-) . . . 286
34.4.3 ESMF_TimeOperator(-) . . . 287
34.4.4 ESMF_TimeOperator(==) . . . 287
34.4.5 ESMF_TimeOperator(/=) . . . 288
34.4.6 ESMF_TimeOperator(<) . . . 288
34.4.7 ESMF_TimeOperator(<=) . . . 289
34.4.8 ESMF_TimeOperator(>) . . . 289
34.4.9 ESMF_TimeOperator(>=) . . . 290
34.4.10 ESMF_TimeGet . . . 290
34.4.11 ESMF_TimeIsSameCalendar . . . 293
34.4.12 ESMF_TimePrint . . . 293
34.4.13 ESMF_TimeSet . . . 294
34.4.14 ESMF_TimeSyncToRealTime . . . 296
34.4.15 ESMF_TimeValidate . . . 296

35 TimeInterval Class **297**
35.1 Description 297
35.2 Use and Examples 297
35.2.1 Time Interval Initialization 298
35.2.2 Time Interval Conversion 298
35.2.3 Time Interval Difference 298
35.2.4 Time Interval Multiplication 298
35.2.5 Time Interval Comparison 299
35.3 Restrictions and Future Work 299
35.4 Class API 299
35.4.1 ESMF_TimeIntervalOperator(+) 299
35.4.2 ESMF_TimeIntervalOperator(-) 300
35.4.3 ESMF_TimeIntervalOperator(/) 300
35.4.4 ESMF_TimeIntervalFunction(MOD) 301
35.4.5 ESMF_TimeIntervalOperator(x) 301
35.4.6 ESMF_TimeIntervalOperator(x) 302
35.4.7 ESMF_TimeIntervalOperator(==) 302
35.4.8 ESMF_TimeIntervalOperator(/=) 303
35.4.9 ESMF_TimeIntervalOperator(<) 303
35.4.10 ESMF_TimeIntervalOperator(<=) 304
35.4.11 ESMF_TimeIntervalOperator(>) 304
35.4.12 ESMF_TimeIntervalOperator(>=) 305
35.4.13 ESMF_TimeIntervalAbsValue 305
35.4.14 ESMF_TimeIntervalGet 306
35.4.15 ESMF_TimeIntervalNegAbsValue 308
35.4.16 ESMF_TimeIntervalPrint 308
35.4.17 ESMF_TimeIntervalSet 309
35.4.18 ESMF_TimeIntervalValidate 311

36 Clock Class **312**
36.1 Description 312
36.2 Use and Examples 312
36.2.1 Clock Creation 313
36.2.2 Clock Advance 313
36.2.3 Clock Examination 313
36.2.4 Clock Destruction 314
36.3 Restrictions and Future Work 314
36.4 Class API 314
36.4.1 ESMF_ClockOperator(==) 314
36.4.2 ESMF_ClockOperator(/=) 315
36.4.3 ESMF_ClockAdvance 315
36.4.4 ESMF_ClockCreate 316
36.4.5 ESMF_ClockCreate 317
36.4.6 ESMF_ClockDestroy 317
36.4.7 ESMF_ClockGet 318
36.4.8 ESMF_ClockGetAlarm 319
36.4.9 ESMF_ClockGetAlarmList 319
36.4.10 ESMF_ClockGetNextTime 320
36.4.11 ESMF_ClockIsStopTime 321
36.4.12 ESMF_ClockPrint 321
36.4.13 ESMF_ClockSet 322

36.4.14 ESMF_ClockSyncToRealTime . . . 323
36.4.15 ESMF_ClockValidate . . . 323

37 Alarm Class **324**
37.1 Description . . . 324
37.2 Alarm Options . . . 324
37.2.1 ESMF_AlarmListType . . . 324
37.3 Use and Examples . . . 324
37.3.1 Clock Initialization . . . 325
37.3.2 Alarm Initialization . . . 325
37.3.3 Clock Advance and Alarm Processing . . . 326
37.3.4 Alarm and Clock Destruction . . . 326
37.4 Restrictions and Future Work . . . 327
37.5 Design and Implementation Notes . . . 327
37.6 Class API . . . 327
37.6.1 ESMF_AlarmOperator(==) . . . 327
37.6.2 ESMF_AlarmOperator(/=) . . . 327
37.6.3 ESMF_AlarmCreate . . . 328
37.6.4 ESMF_AlarmCreate . . . 329
37.6.5 ESMF_AlarmDestroy . . . 330
37.6.6 ESMF_AlarmDisable . . . 330
37.6.7 ESMF_AlarmEnable . . . 331
37.6.8 ESMF_AlarmGet . . . 331
37.6.9 ESMF_AlarmIsEnabled . . . 332
37.6.10 ESMF_AlarmIsRinging . . . 333
37.6.11 ESMF_AlarmIsSticky . . . 333
37.6.12 ESMF_AlarmNotSticky . . . 334
37.6.13 ESMF_AlarmPrint . . . 334
37.6.14 ESMF_AlarmRingerOff . . . 335
37.6.15 ESMF_AlarmRingerOn . . . 335
37.6.16 ESMF_AlarmSet . . . 336
37.6.17 ESMF_AlarmSticky . . . 337
37.6.18 ESMF_AlarmValidate . . . 337
37.6.19 ESMF_AlarmWasPrevRinging . . . 338
37.6.20 ESMF_AlarmWillRingNext . . . 338

38 Config Class **339**
38.1 Description . . . 339
38.2 Use and Examples . . . 339
38.2.1 Resource Files . . . 339
38.2.2 A Quick Overview . . . 339
38.2.3 Package History . . . 340
38.3 Class API . . . 341
38.3.1 ESMF_ConfigCreate . . . 341
38.3.2 ESMF_ConfigDestroy . . . 341
38.3.3 ESMF_ConfigFindLabel . . . 341
38.3.4 ESMF_ConfigGetAttribute . . . 342
38.3.5 ESMF_ConfigGetAttribute . . . 342
38.3.6 ESMF_ConfigGetAttribute . . . 343
38.3.7 ESMF_ConfigGetAttribute . . . 344
38.3.8 ESMF_ConfigGetAttribute . . . 344

38.3.9 ESMF_ConfigGetAttribute . . . 345
38.3.10 ESMF_ConfigGetAttribute . . . 345
38.3.11 ESMF_ConfigGetAttribute . . . 346
38.3.12 ESMF_ConfigGetAttribute . . . 347
38.3.13 ESMF_ConfigGetChar . . . 347
38.3.14 ESMF_ConfigGetDim . . . 348
38.3.15 ESMF_ConfigGetLen . . . 348
38.3.16 ESMF_ConfigLoadFile . . . 349
38.3.17 ESMF_ConfigNextLine . . . 349

39 LogErr Class 350
39.1 Description . . . 350
39.2 LogErr Options . . . 350
39.2.1 ESMF_MsgType . . . 350
39.2.2 ESMF_LogType . . . 350
39.3 Use and Examples . . . 350
39.3.1 Default Log . . . 351
39.3.2 User Created Log . . . 352
39.4 Restrictions and Future Work . . . 352
39.5 Design and Implementation Notes . . . 353
39.6 Object Model . . . 353
39.7 Class API . . . 354
39.7.1 ESMF_LogClose . . . 354
39.7.2 ESMF_LogFoundAllocError . . . 354
39.7.3 ESMF_LogFoundError . . . 355
39.7.4 ESMF_LogMsgFoundAllocError . . . 355
39.7.5 ESMF_LogMsgFoundError . . . 356
39.7.6 ESMF_LogOpen . . . 357
39.7.7 ESMF_LogWrite . . . 357

40 DELayout Class 358
40.1 Description . . . 358
40.2 Use and Examples . . . 359
40.2.1 Default 1-D DELayout . . . 359
40.2.2 1-D DELayout with Fixed Number of DEs . . . 359
40.2.3 2-D DELayout with Fixed Number of DEs . . . 360
40.3 Restrictions and Future Work . . . 360
40.4 Design and Implementation Notes . . . 360
40.5 Class API . . . 360
40.5.1 ESMF_DELayoutCreate . . . 360
40.5.2 ESMF_DELayoutDestroy . . . 361
40.5.3 ESMF_DELayoutGet . . . 362
40.5.4 ESMF_DELayoutGetDELocalInfo . . . 362
40.5.5 ESMF_DELayoutGetDEMatch . . . 363
40.5.6 ESMF_DELayoutPrint . . . 364

41 VM Class 364
41.1 Description . . . 364
41.2 Use and Examples . . . 365
41.2.1 VM Default Basics Example . . . 365
41.2.2 VMGet MPI Communicator Example . . . 366

41.2.3 VMSend/VMRecv Example . . . 366
41.2.4 VMScatter/VMGather Example . . . 366
41.2.5 VMAllFullReduce Example . . . 367
41.2.6 VM Component Example . . . 367
41.3 Restrictions and Future Work . . . 368
41.4 Design and Implementation Notes . . . 369
41.5 Class API . . . 371
41.5.1 ESMF_VMAllFullReduce . . . 371
41.5.2 ESMF_VMAllFullReduce . . . 372
41.5.3 ESMF_VMAllFullReduce . . . 373
41.5.4 ESMF_VMAllReduce . . . 374
41.5.5 ESMF_VMAllReduce . . . 375
41.5.6 ESMF_VMAllReduce . . . 375
41.5.7 ESMF_VMBarrier . . . 376
41.5.8 ESMF_VMGather . . . 377
41.5.9 ESMF_VMGather . . . 377
41.5.10 ESMF_VMGather . . . 378
41.5.11 ESMF_VMGet . . . 379
41.5.12 ESMF_VMGetGlobal . . . 380
41.5.13 ESMF_VMGetPETLocalInfo . . . 380
41.5.14 ESMF_VMPrint . . . 381
41.5.15 ESMF_VMRecv . . . 381
41.5.16 ESMF_VMRecv . . . 382
41.5.17 ESMF_VMRecv . . . 383
41.5.18 ESMF_VMScatter . . . 383
41.5.19 ESMF_VMScatter . . . 384
41.5.20 ESMF_VMScatter . . . 385
41.5.21 ESMF_VMSend . . . 386
41.5.22 ESMF_VMSend . . . 386
41.5.23 ESMF_VMSend . . . 387
41.5.24 ESMF_VMSendRecv . . . 388
41.5.25 ESMF_VMSendRecv . . . 388
41.5.26 ESMF_VMSendRecv . . . 389
41.5.27 ESMF_VMThreadBarrier . . . 390
41.5.28 ESMF_VMWait . . . 391

42 Bibliography **392**

Bibliography **392**

V Appendices **393**

43 Appendix A: A Brief Introduction to UML **393**

Part I
ESMF Overview

1 Release Notes

ESMF v2.0 is a first usable release of the Earth System Modeling Framework. While the ESMF still has much growing to do over the coming years, we expect modelers to find in this release tools that benefit real codes. You may choose to start with the highest level of functionality in the framework, the software for representing models as components and coupling them to other models; or the lowest level, the toolkits for data communication, I/O, logging, or calendar management. Wherever you begin, we hope that you find the ESMF useful, and look forward to hearing your comments on any aspect of the software. Section 4 of this document includes instructions on submitting comments on ESMF to our development team.

2 What is the Earth System Modeling Framework?

The ESMF is a structured collection of software building blocks that can be used or customized to develop Earth system model components, and assemble them into applications. The simplest view of the ESMF is that it consists of an **infrastructure** of utilities and data structures for creating model components, and a **superstructure** for coupling them. User code sits between these two layers, making calls to the infrastructure libraries beneath it and being scheduled and synchronized by the superstructure above it. The configuration resembles a sandwich, as shown in Figure 1.

The ESMF architecture is scalable, flexible paradigm for building highly complex climate, weather, and related applications from components such as atmospheric models, land models, and data assimilation systems. The ESMF is not a single master application into which all components must fit; rather it is a way of developing components so that they can be used in many different user-written applications. Model components that adopt ESMF are designed to be usable in different contexts without code modification, and may be incorporated into other ESMF-based modeling systems within the Earth science community. In addition to high-level organization, ESMF provides a set of robust, portable, performance optimized libraries for regridding, data transfers, I/O, time management, and other common modeling functions. ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap user-written components in ESMF interfaces in order to adopt the ESMF architecture and utilize framework coupling services.

3 The ESMF Reference Manual for Fortran

The ESMF provides both Fortran and C++ versions of its interfaces for many methods. This *ESMF Reference Manual* is a listing of ESMF standard interfaces for Fortran.[1]

Interfaces are grouped by class. A class is an object-oriented software design construct that embodies a specific concept like a physical field. Superstructure classes are listed first in this *Manual*, followed by infrastructure classes.

The major classes in the ESMF superstructure are Components, which typically represent large pieces of functionality such as models, model couplers, and dynamics and physics packages; and States, which are the data structures used to store the fields and other data Components require or can make available. There are both data structures and utilities in the ESMF infrastructure; classes include Fields, collections of Fields on the same grid (called Bundles), Arrays, and utilities for communication, decomposition, time management, and application configuration.

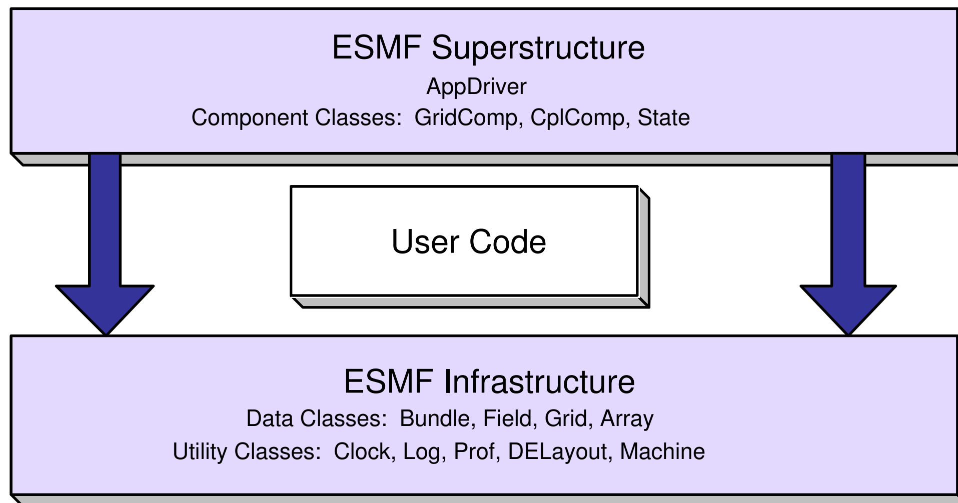
For how to get started with ESMF, see the *ESMF User's Guide*. This document includes installation instructions, an overview of the whole framework, an extended example of a coupled code, and other useful information.

4 How to Contact User Support and Find Additional Information

The ESMF team can answer questions about the interfaces presented in this document. For user support, please contact esmf_support@ucar.edu.

[1] Since the audience for it is small, we have not yet prepared a comprehensive reference manual for C++.

Figure 1: Schematic of the ESMF "sandwich" architecture. In this design the framework consists of two parts, an upper level **superstructure** layer and a lower-level **infrastructure** layer. User code is sandwiched between these two layers.



More information on the ESMF project as a whole is available on the ESMF website, http://www.esmf.ucar.edu. The website includes a description of ESMF testbed applications, related projects, the ESMF management structure, and more. The *ESMF User's Guide* contains installation instructions, an overview of the ESMF system and a description of how its classes interrelate. Other documents available on the ESMF site include an exhaustive *ESMF Requirements Document* and an *ESMF Developer's Guide* that details our project procedures and conventions.

5 How to Submit New Requirements

The **Development** link on the ESMF website includes on-line forms for the submission of new requirements, if it seems that the current API does not satisfy the needs of your application. We welcome input on any aspect of the ESMF project; general questions and comments should be sent to esmf@ucar.edu.

6 Conventions

6.1 Document Conventions

The following conventions for fonts and capitalization are used in this document.

Style	Meaning	Example
italics	documents	*ESMF Reference Manual*
`courier`	code fragments	`ESMF_TRUE`
`courier()`	ESMF method name	`ESMF_FieldGet()`
boldface	first definitions	An **address space** is ...
boldface	web links	**Development** webpage
Capitals	ESMF class name	DataMap

ESMF class names frequently coincide with words commonly used within the Earth system domain (field, grid, component, array, etc.) The convention we adopt in this manual is that if a word is used in the context of an ESMF

class name it is capitalized, and if the word is used in a more general context it remains in lower case. We would write, for example, that an ESMF Field class represents a physical field.

Diagrams are drawn using the Unified Modeling Language (UML). UML is a visual tool that can illustrate the structure of classes, define relationships between classes, and describe sequences of actions. A reader interested in more detail can refer to a text such as *The Unified Modeling Language Reference Manual.* [3]

6.2 Method Name and Argument Conventions

There are conventions for how class methods are presented throughout this document. Although Fortran interfaces are not case-sensitive, we use case to help parse multi-word names. We also use case to help make the presentation of Fortran interfaces consistent with the presentation of C++ interfaces.

Method names begin with ESMF_, followed by the class name, followed by the name of the operation being performed. Each new word is capitalized.

For method arguments that are multi-word, the first word is lower case and subsequent words begin with upper case. ESMF class names (including typed flags) are an exception. When multi-word class names appear in argument lists, all letters after the first are lower case. The first letter is lower case if the class is the first word in the argument and upper case otherwise. For example, in an argument list the DELayout class name may appear as delayout or srcDelayout.

Most Fortran calls in the ESMF are subroutines, with any returned values passed through the interface. For the sake of convenience, some ESMF calls are written as functions.

A typical ESMF call thus looks like this:

```
call ESMF_<ClassName><Operation>(classname, firstArgument,
          secondArgument, ..., rc)
```

where
<ClassName> is the class name,
<method> is the name of the action to be performed,
classname is a variable of the derived type associated with the class,
the arg* arguments are whatever other variables are required for the operation,
and rc is a return code.

6.3 Locating Methods in this Manual

Methods for each class are located in the section devoted to that class in the *Reference Manual*. In some classes, methods are split into a number of different types. For example, there are separate listings for Basic Field Methods, Field Overloads for Fortran Arrays, and Field Communications. The methods in each listing are ordered alphabetically. The split into different listings is a side effect of the automated document generation system we use; it reflects which methods are located in the same source files. It is something we are working to eliminate!

7 The ESMF Application Programming Interface

The ESMF Application Programming Interface (API) is based on the object-oriented programming notion of a **class**. A class is a software construct that's used for grouping a set of related variables together with the subroutines and functions that operate on them. We use classes in ESMF because they help to organize the code, and often make it easier to maintain and understand. A particular instance of a class is called an **object**. For example, Field is an ESMF class. An actual Field called temperature is an object. That is about as far as we will go into formal software engineering terminology.

The Fortran interface is implemented so that the variables associated with a class are stored in a derived type. For example, an ESMF_Field derived type stores the data array, grid information, and metadata associated with a physical field. The derived type for each class is stored in a Fortran module, and the operations associated with each class are defined as module procedures. We use the Fortran features of generic functions and optional arguments extensively to simplify our interfaces.

The modules for ESMF are bundled together and can be accessed with a single USE statement, USE ESMF_Mod.

7.1 Standard Methods and Interface Rules

ESMF defines a set of standard methods and interface rules that hold across the entire API. These are:

- ESMF_<Class>Create() and ESMF_<Class>Destroy(), for creating and destroying classes. The ESMF_<Class>Create() method allocates memory for the class structure itself and for internal variables, and initializes variables as appropriate. It is always written as a Fortran function that returns a derived type instance of the class.
- ESMF_<Class>Set() and ESMF_<Class>Get(), for setting and retrieving a particular item or flag. In general, these methods are overloaded for all cases where the item can be manipulated as a name/value pair. If identifying the item requires more than a name, or if the class is of sufficient complexity that overloading in this way would result in an overwhelming number of options, we define specific ESMF_<Class>Set<Something>() and ESMF_<Class>Get<Something>() interfaces.
- ESMF_<Class>Add(), ESMF_<Class>Get(), and ESMF_<Class>Remove() for manipulating items that can be appended or inserted into a list of like items within a class. For example, the ESMF_StateAddField() method adds another Field to the list of Fields contained in the State class.
- ESMF_<Class>Print(), for printing the contents of a class to standard out. This method is mainly intended for debugging.
- ESMF_<Class>ReadRestart() and ESMF_<Class>WriteRestart(), for saving the contents of a class and restoring it exactly. Read and write restart methods have not yet been implemented for most ESMF classes, so where necessary the user needs to write restart values themselves.
- ESMF_<Class>Validate(), for determining whether a class is internally consistent. For example, ESMF_FieldValidate checks whether the Array and Grid associated with a Field are consistent.

EXAMPLE
In this simple example, an ESMF Field is created with the name 'temp'.

```
USE ESMF_Mod

type ESMF_Field :: field

field = ESMF_FieldCreate('temp')
```

7.2 Deep and Shallow Classes

The ESMF contains two types of classes. **Deep** classes require ESMF_<Class>Create() and ESMF_<Class>Destroy() calls. They take significant time to set up and should not be created in a time-critical portion of code. Deep objects persist even after the method in which they were created has returned. Most classes in the ESMF, including Fields, Bundles, Arrays, Grids and Clocks, fall into this category.

Shallow classes do not require ESMF_<Class>Create() and ESMF_<Class>Destroy() calls. They can simply be declared and their values set using an ESMF_<Class>Set() call. Shallow classes do not take long to set up and can be declared and set within a time-critical code segment. Shallow objects stop existing when the method in which they were declared has returned.

An exception to this is when a shallow object, such as an IOSpec, is used to carry values into a deep object, for example during an ESMF_FieldCreate() call during an application initialization phase. In this case an IOSpec is passed in through the ESMF_FieldCreate() argument list and the values of the IOSpec are copied into the new Field object. Although the IOSpec is destroyed when the initialization phase ends, the Field carries a copy of the IOSpec in persistent memory. This internal IOSpec is destroyed with the ESMF_FieldDestroy() call.

Other examples of shallow classes are Times, TimeIntervals, and ArraySpecs.

See Section 11, Overall Design and Implementation Notes, for a brief discussion of deep and shallow classes from an implementation perspective. For an in-depth look at the design and inter-language issues related to deep and shallow classes, please see the ESMF Implementation Report.

7.3 Special Methods

The following are special methods which, in one case, are required by any application using ESMF, and in the other case must be called by any application that is using ESMF Components.

- ESMF_Initialize() and ESMF_Finalize() are required methods that must bracket the use of ESMF within an application. They manage the resources required to run ESMF and shut it down gracefully.

- ESMF_<Type>CompInitialize(), ESMF_<Type>CompRun(), and ESMF_<Type>CompFinalize() are component methods that are used at the highest level within ESMF. <Type> may be <Grid>, for Gridded Components such as oceans or atmospheres, or <Cpl>, for Coupler Components that are used to connect them. The content of these methods is not part of the ESMF. Instead the methods call into associated Fortran subroutines within user code.

7.4 The ESMF Data Hierarchy

The ESMF API is organized around an hierarchy of five classes that contain model field data. The operations that are performed on model field data, such as regridding, redistribution, and halo updates, are accessed through these classes.

The main data classes in ESMF, in order of increasing complexity, are:

- **Array** An ESMF Array is a distributed, multi-dimensional array that can carry information such as its type, kind, rank, and associated halo widths. It contains a reference to a native Fortran array.

- **Field** A Field represents a physical scalar or vector field. It contains a reference to an Array along with grid information and metadata.

- **Bundle** A Bundle is a collection of Fields discretized on the same grid. The staggering of data points may be different for different Fields within a Bundle.

- **State** A State represents the collection of data that a Component either requires to run (an Import State) or can make available to other Components (an Export State). States may contain references to Bundles, Fields, or Arrays.

- **Component** A Component is a substantial piece of software with a distinct function. ESMF currently recognizes two types of Components. Components that represent a physical domain or process, such as an atmospheric model, are called Gridded Components since they are usually discretized on an underlying grid. The Components responsible for regridding and transferring data between Gridded Components are called Coupler Components. Each Component is associated with an Import and an Export State. Components can be nested so that simpler components and applications can be used to compose more complex applications.

Underlying these data classes are native language arrays. ESMF allows you to reference an existing Fortran array to an ESMF Array, Field, or Bundle, so that ESMF data classes can be readily introduced into existing code. You can perform communication operations directly on Fortran arrays through the DELayout class, which serves as a unifying wrapper for distributed and shared memory communication libraries.

7.5 ESMF Spatial Classes

Like the hierarchy of model data classes, ranging from the simple to the complex, the ESMF is organized around an hierarchy of classes that represent different spaces associated with a computation. Each of these spaces can be indexed in some fashion, in order to give the user control over how a computation is executed. For Earth system applications, this hierarchy spans the environment associated with the computer to the physical region described by the application. The main spatial classes in ESMF, in order of those closest to the machine to those closest to the application, are:

- The **Virtual Machine**, or **VM** The ESMF VM is an abstraction of a parallel computing environment that encompasses both shared and distributed memory. Its primary purpose is resource allocation. Each Component defines its own VM based on the resources it desires. The elements of a VM are **Persistent Execution Threads**, or **PETs**. A simple case is one in which every PET is associated with an MPI process running on a separate processor. If Components are nested, the parent component allocates a subset of its PETs to its children. The children have some flexibility, subject to the constraints of the computing environment, to decide how they want to use the PETs they've received.
- **DELayout** A DELayout represents a decomposition. Its basic elements are **Decomposition Elements**, or **DEs**. A DELayout associates a set of DEs and a topology - how the DEs are connected - with the PETs in a VM. The user can also define communication weights between DEs, for use in load balancing. DEs are not necessarily one-to-one with PETs. For cache blocking, or user-managed multi-threading, more DEs than PETs may be defined. Fewer DEs than PETs may be defined if an application requires, for example, a decomposition that is an integer multiple.
- **Grid** A Grid is an abstraction of a physical space. It associates a coordinate system, a set of coordinates, and a topology to a collection of grid cells.
- **Field** A Field may contain more dimensions than the Grid that it is discretized on. For example, for convenience during integration, a user may want to define a single Field object that holds snapshots of the data at multiple times. The Field must track what these additional dimensions mean. Fields also keep track of the location of a Field data point within its associated Grid cell.

Although it is not an ESMF class, the linear **address space** of the computer is another fundamental index space that must be mapped to data stored by the ESMF system.

7.6 ESMF DataMap Classes

In order to map the index spaces of the spatial classes, we require either implicit rules (in which case the relationship between index spaces is defined by default), or special classes that allow the user to specify the desired association. The following classes define how the data is laid out in memory.

- **ArrayDataMap** The ArrayDataMap class specifies how the address space of the computer relates to the array rank (e.g. row or column major order), and, optionally, how a list of array ranks corresponds to a list of Grid dimensions.
- **FieldDataMap** The FieldDataMap specifies the number of directional components in a vector Field, and how they are interleaved.
- **BundleDataMap** The BundleDataMap dictates how the Fields within a Bundle are interleaved.

7.7 ESMF Specification Classes

At various places in the ESMF, it is useful to make neat packets of descriptive parameters. Some of these are:

- **IOSpec**, for storing IO parameters.
- **ArraySpec**, for storing the specifics, such as type/kind/rank, of an array.

7.8 ESMF Utility Classes

There are a number of utilities in ESMF that can be used independently. These are:

- **TimeMgr**, for calendar, date, clock and alarm functions.
- **LogErr**, for logging and error handling.
- **Config**, for creating resource files that can replace namelists as a consistent way of setting configuration parameters.

8 Overall Rules and Behavior

8.1 Allocation Rules

The basic rule of allocation and deallocation for the ESMF is: whoever allocates it is responsible for deallocating it.

ESMF methods that allocate their own space for data will deallocate that space when the object is destroyed. Methods which accept a user-allocated buffer, for example `ESMF_FieldCreate()` with the `ESMF_DATA_REF` flag, will not deallocate that buffer at the time the object is destroyed. The user must arrange for the buffer to be deallocated when all use of it is complete.

Classes such as Fields, Bundles, and States may have Arrays, Fields, Grids and Bundles created externally and associated with them. These associated items are not destroyed along with the rest of the data object since it is possible for the items to be added to more than one data object at a time (e.g. the same Grid could be part of many Fields). It is the user's responsibility to delete these items when the last use of them is done.

8.2 Attributes

Attributes are (name, value) pairs, where the name is a character string and the value can be either a single value or list of `int/I*4`, `double/R*8`, logical (`ESMF_Logical`), or `char */character` values. Attributes can be associated with Fields, Bundles, and States. Mixed types are not allowed in a single attribute, and all attribute names must be unique within a single object. Attributes are set by name, and can be retrieved either directly by name or by querying for a count of attributes and retrieving names and values by index number.

9 Integrating ESMF into Applications

Depending on the requirements of the application, the user may want to begin integrating ESMF in either a top-down or bottom-up manner. In the top-down approach, tools at the superstructure level are used to help reorganize and structure the interactions among large-scale components in the application. It is appropriate when interoperability is a primary concern; for example, when several different versions or implementations of components are going to be swapped in, or a particular component is going to be used in multiple contexts. Another reason for deciding on a top-down approach is that the application contains legacy code that for some reason (e.g., very large, difficult to work with, highly performance-tuned, resource limitations) there is little motivation to fully restructure. The superstructure can be incorporated into such applications in a way that is non-intrusive.

In the bottom-up approach, the user selects desired utilities (data communications, calendar management, performance profiling, logging and error handling, etc.) from the ESMF infrastructure and either writes new code using them, introduces them into existing code, or replaces the functionality in existing code with them. This makes sense when there is a specific need for some functionality, like robust data acommunications, or when the component writer is starting from scratch.

9.1 Using the ESMF Superstructure

The following is a typical set of steps involved in adopting the ESMF superstructure. The first two tasks, which occur before an ESMF call is ever made, have the potential to be the most difficult and time-consuming. They are the work of splitting an application into components and ensuring that each component has well-defined stages of execution.[2]

1. Decide how to organize the application as discrete Gridded and Coupler Components. The developer might need to reorganize code so that individual components are cleanly separated and their interactions consist of a minimal number of data exchanges.
2. Divide the code for each component into initialize, run, and finalize methods. These methods can be multi-phase, e.g., `init_1, init_2`.
3. Pack any data that will be transferred between components into ESMF Import and Export State data structures. The user must describe the distribution of grids over resources on a parallel computer via the VM and DELayout.
4. Pack time information into ESMF time management data structures.
5. Using code templates provided in the ESMF distribution, create ESMF Gridded and Coupler Components to represent each component in the user code.
6. Write a set services routine that sets ESMF entry points for each user component's initialize, run, and finalize methods.
7. Run the application using an ESMF Application Driver.

9.2 Using the ESMF Infrastructure

Adoption of infrastructure utilities and data structures can follow many different paths. The calendar management utility is a popular place to start, since there is enough functionality in the ESMF time manager to merit the effort required to integrate it into codes and bundle it with an application.

[2]ESMF aside, this sort of code structure helps to promote application clarity and maintainability, and the effort put into it is likely to be a good investment in any case.

10 Global Options and Parameters

10.1 Flags

10.1.1 ESMF_AllocFlag

DESCRIPTION:
Indicates whether to allocate data or not.
Valid values are:

ESMF_ALLOC Allocate data.

ESMF_NO_ALLOC Do not allocate data at this time.

10.1.2 ESMF_BlockingFlag

DESCRIPTION:
Indicates whether to block during a communication call.
Valid values are:

ESMF_BLOCKING The called method will block until all (PET-)local operations are complete. After the return of a blocking method it is safe to modify or use all participating local data.

ESMF_NONBLOCKING The called method will not block but return immediately after initiating the requested operation. It is unsafe to modify or use participating local data before all local operations have completed.

10.1.3 ESMF_CopyFlag

DESCRIPTION:
Indicates whether to reference a data item or make a copy of it.
Valid values are:

ESMF_DATA_COPY Copy the data item to another buffer.

ESMF_DATA_REF Reference the data item.

10.1.4 ESMF_IndexFlag

DESCRIPTION:
Indicates whether index is local (per DE) or global (per object).
Valid values are:

ESMF_INDEX_DELOCAL Refers to indices on the local DE.

ESMF_INDEX_GLOBAL Refers to object-wide indices.

10.1.5 ESMF_InterleaveFlag

DESCRIPTION:
Interleave is used when there are multiple variables or if individual data items are vectors. Used in `ESMF_FieldDataMap` and `ESMF_BundleDataMap`. (The interleave option is not yet implemented.)
Valid values are:

ESMF_INTERLEAVE_BY_BLOCK Items are listed in blocks, all items of one type followed by all items of the next type.

ESMF_INTERLEAVE_BY_ITEM Items are interleaved item by item.

10.1.6 ESMF_NeededFlag

DESCRIPTION:
Specifies whether or not a data item is needed for a particular application configuration. Used in `ESMF_State`.
Valid values are:

ESMF_NEEDED Data is needed.

ESMF_NOTNEEDED Data is not needed.

10.1.7 ESMF_ReadyFlag

DESCRIPTION:
Specifies whether a data item is ready to read or write.
Valid values are:

ESMF_READYTOREAD Data is ready to read.

ESMF_READYTOWRITE Data is ready to write.

ESMF_NOTREADY Data is not ready.

10.1.8 ESMF_ReduceFlag

DESCRIPTION:
Indicates reduce operation to a `Reduce()` method.
Valid values are:

ESMF_SUM Use arithmetic sum to add all data elements.

ESMF_MIN Determine the minimum of all data elements.

ESMF_MAX Determine the maximum of all data elements.

10.1.9 ESMF_ReqForRestartFlag

DESCRIPTION:
Specifies whether a data item is necessary for restart.
Valid values are:

ESMF_REQUIRED_FOR_RESTART Data is required for restart.

ESMF_NOTREQUIRED_FOR_RESTART Data is not required for restart.

10.1.10 ESMF_ValidFlag

DESCRIPTION:
Specifies whether a data item contains valid data.
Valid values are:

ESMF_VALID Data is ready to read.

ESMF_INVALID Data is ready to write.

ESMF_NOTREADY Data is not ready.

10.2 Parameters

10.2.1 ESMF_DataKind

DESCRIPTION:
Supported ESMF data kinds.
Valid values are:

ESMF_I1 1 byte integer.

ESMF_I2 2 byte integer.

ESMF_I4 4 byte integer.

ESMF_I8 8 byte integer.

ESMF_R4 4 byte real.

ESMF_R8 8 byte real.

ESMF_C8 8 byte character.

ESMF_C16 16 byte character.

10.2.2 ESMF_DataType

DESCRIPTION:
Supported ESMF data types.
Valid values are:

ESMF_DATA_INTEGER Integer type.

ESMF_DATA_REAL Real type.

ESMF_DATA_LOGICAL Logical type.

ESMF_DATA_CHARACTER Character type.

11 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 7.2 differ in how and where they are allocated within a multi-language implementation environment. We distinguish between the implementation language, which is the language a method is written in, and the calling language, which is the language that the user application is written in. Deep classes are allocated off the process heap by the implementation language. Shallow classes are allocated off the stack by the calling language.

2. **Base class.** All ESMF classes are built upon a Base class. The Base is used to hold system-wide capabilities, such as Attributes. Attributes are implemented in the Base class so they can be attached to any object in the system which is built on the Base object. (This is true for all deep objects in the system.) Attributes are created by making a private copy of the information provided during the Set call. Lists of values are supported, but they are not intended for large data arrays. Attribute data is copied during a Get operation.

Part II
Superstructure

12 Overview of Superstructure

ESMF superstructure classes define an architecture for assembling Earth system applications from modeling **components**. A component may be defined in terms of the physical domain that it represents, such as an atmosphere or sea ice model. It may also be defined in terms of a computational function, such as a data assimilation system. Earth system research often requires that such components be **coupled** together to create an application. By coupling we mean the data transformations and, on parallel computing systems, data transfers, that are necessary to allow data from one component to be utilized by another. ESMF offers regridding methods and other tools to simplify the organization and execution of inter-component data exchanges.

In addition to components defined at the level of major physical domains and computational functions, components may be defined that represent smaller computational functions within larger components, such as the transformation of data between the physics and dynamics in a spectral atmosphere model, or the creation of nested higher resolution regions within a coarser grid. The objective is to couple components at varying scales both flexibly and efficiently. ESMF encourages a hierachical application structure, in which large components branch into smaller sub-components (see Figure 2). ESMF also makes it easier for the same component to be used in multiple contexts without changes to its source code.

Key Features
Modular, component-based architecture.
Hierarchical assembly of components into applications.
Use of components in multiple contexts without modification.
Sequential or concurrent component execution.
Single program, multiple datastream (SPMD) applications for maximum portability and reconfigurability.

12.1 Superstructure Classes

There are a small number of classes in the ESMF superstructure:

- **Component** An ESMF component has two parts, one that is supplied by the ESMF and one that is supplied by the user. The part that is supplied by the framework is an ESMF derived type that is either a Gridded Component (**GridComp**) or a Coupler Component (**CplComp**). A Gridded Component typically represents a physical domain in which data is associated with one or more grids - for example, a sea ice model. A Coupler Component arranges and executes data transformations and transfers between one or more Gridded Components. Gridded Components and Coupler Components have standard methods, which include initialize, run, and finalize. These methods can be multi-phase.

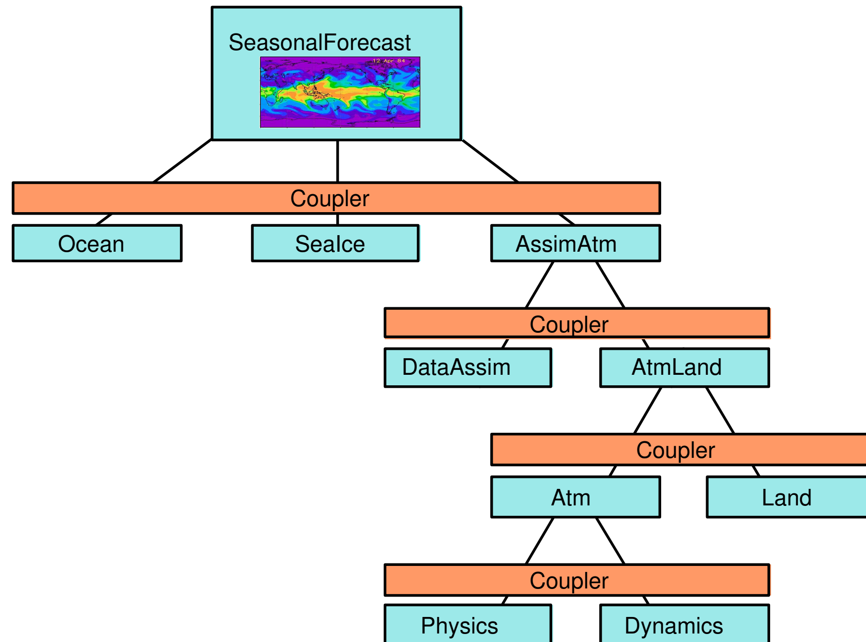
  The second part of an ESMF Component is user code, such as a model or data assimilation system. Users set entry points within their code so that it is callable by the framework. In practice, setting entry points means that within user code there are calls to ESMF methods that associate the name of a Fortran subroutine with a corresponding standard ESMF operation. For example, a user-written initialization routine called `popOceanInit` might be associated with the standard Initialize routine of an ESMF Gridded Component named "POP" that represents an ocean model.

- **State** ESMF components exchange information with other components only through States. A State is an ESMF derived type that can contain Fields, Bundles, Arrays, and other States. A Gridded Component is associated with two States, an **Import State** and an **Export State**. Its Import State holds the data that it receives from other Gridded Components. Its Export State contains data that it can make available to other Gridded Components.

- **Application Driver** The Application Driver (**AppDriver**) is a small, generic driver program that contains the "main" routine for an ESMF application.

An ESMF coupled application typically involves an AppDriver, a parent Gridded Component, two or more child Gridded Components that require an inter-component data exchange, and one or more Coupler Components.

Figure 2: ESMF enables applications such as a seasonal forecast model to be structured hierarchically, and reconfigured and extended easily.



The parent Gridded Component is responsible for creating the child Gridded Components that are exchanging data and creating the Coupler, for creating the necessary Import and Export States, and for setting up the desired sequencing. The AppDriver "main" routine calls the parent Gridded Component's initialize, run, and finalize methods in order to execute the application. For each of these standard methods, the parent Gridded Component in turn calls the corresponding methods in the child Gridded Components and the Coupler Component. For example, consider a simple coupled ocean/atmosphere simulation. When the initialize method of the parent Gridded Component is called by the AppDriver, it in turn calls the initialize methods of its child atmosphere and ocean Gridded Components, and the initialize method of an ocean-to-atmosphere Coupler Component. Figure 3 shows this schematically for a coupled hurricane model with ocean and atmosphere components.

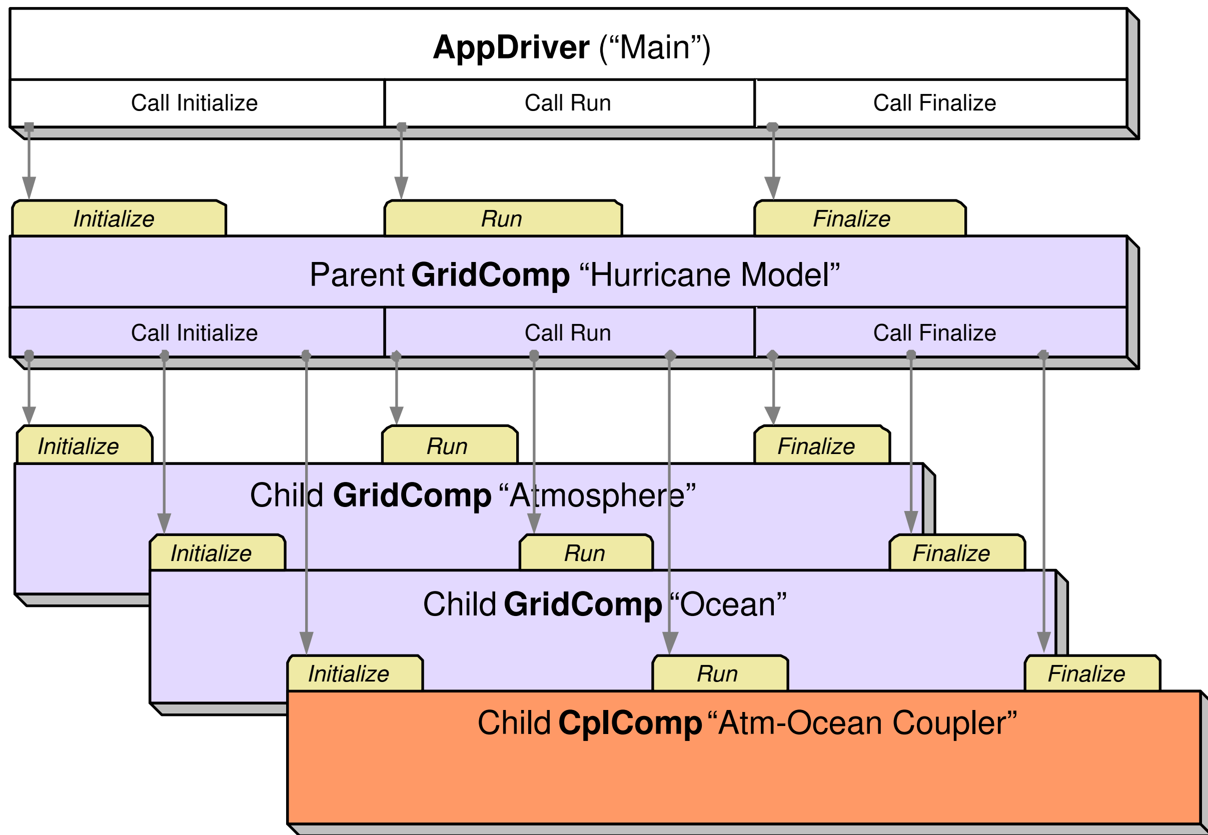
12.2 Distribution and Scoping of Components

Components are allocated computational resources in the form of **Persistent Execution Threads**, or **PET**s. A list of a Component's PETs is contained in a structure called a **Virtual Machine**, or **VM**. The VM also contains information about the topology and characteristics of the underlying computer.

All data transfers within an ESMF application occur *within* a component. For example, a Gridded Component may contain halo updates. Another example is that a Coupler Component may contain a regridding and data redistribution between two Gridded Components. As a result, the architecture of ESMF does not depend on any particular data communication mechanism, and new communication schemes can be introduced without affecting the overall structure of the application.

Since all data communication happens within a component, a Coupler Component must be created on the union of the PETs of all the Gridded Components that it couples.

Figure 3: A call to a standard ESMF initialize (run, finalize) method by a parent component triggers calls to initialize (run, finalize) all of its child components.



A Gridded Component may exist across all the PETs in an application. A Gridded Component may also reside on a subset of PETs in an application. These PETs may wholly coincide with, be wholly contained within, or wholly contain another component.

When a set of Gridded Components and a Coupler runs in sequence on the same set of PETs the application is executing in a **sequential** mode. When Gridded Components are created and run on mutually exclusive sets of PETs, and are coupled by a Coupler Component that extends over the union of these sets, the mode of execution is **concurrent**.

It is possible for ESMF applications to contain some component sets that are executing sequentially and others that are executing concurrently. We might have, for example, atmosphere and land components created on the same subset of PETs, ocean and sea ice components created on the remainder of PETs, and a Coupler created across all the PETs in the application.

12.3 Performance

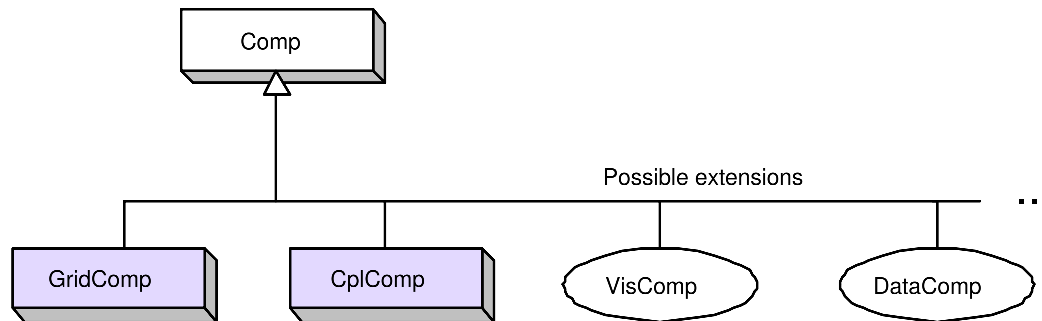
The ESMF design enables the user to configure ESMF applications so that data is transferred directly from one component to another, without requiring that it be copied or sent to a different data buffer as an interim step. This is likely

to be the most efficient way of performing inter-component coupling. However, if desired, an application can also be configured so that data from a source component is sent to a distinct set of Coupler Component PETs for processing before being sent to its destination.

The ability to overlap computation with communication is essential for performance. When running with ESMF the user can initiate data sends during Gridded Component execution, as soon as the data is ready. Computations can then proceed simultaneously with the data transfer.

12.4 Object Model

The following is a simplified UML diagram showing the relationships among ESMF superstructure classes. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



13 Application Driver and Required ESMF Methods

13.1 Description

The ESMF Application Driver (`ESMF_AppDriver`), is a generic ESMF driver program that contains a "main." Simpler applications may be able to use an Application Driver without modification; for more complex applications, an Application Driver can be used as an extendable template.

ESMF provides a number of different Application Drivers in the `$ESMF_DIR/src/Superstructure/AppDriver` directory. An appropriate one can be chosen depending on how the application is to be structured. Options when deciding how to structure an application include choices about:

Sequential vs. Concurrent Execution In a serial execution model every PET executes the same Gridded Component code until it has produced data needed by another Gridded Component, and then all PETs change to running the next Gridded Component or Coupler Component. This has the appeal of simplicity of data consumption and production: when a Gridded Component starts all required data is available for use, and when a Gridded Component finishes all data produced is ready for consumption by the next Gridded Component. This approach also has the possibility of less data movement if the gridding and data decomposition is done such that each processor's memory contains the data needed by the next Component.

In a concurrent execution model subgroups of PETs run Gridded Components and all Gridded Components are active at the same time. Data exchange must be coordinated between Gridded Components so that data deadlock does not occur. This strategy has the advantage of allowing coupling to other Gridded Components at any time during the computational process, including not having to return to the calling level of code before making data available. ESMF does not fully support the concurrent mode of execution at this time.

Pairwise vs. Hub and Spoke Coupler Components are responsible for taking data from one Gridded Component and putting it into the form expected by another Gridded Component. This might include regridding, change of units, averaging, or binning.

Coupler Components can be written for *pairwise* data exchange: the Coupler Component takes data from a single Component and transforms it for use by another single Gridded Component. This simplifies the structure of the Coupler Component code.

Couplers can also be written using a *hub and spoke* model where a single Coupler accepts data from all other Components, can do data merging or splitting, and formats data for all other Components.

Multiple Couplers, using either of the above two models or some mixture of these approaches, are also possible.

Implementation Language The ESMF framework is implemented with a set of Fortran and C++ interfaces to all functions. The main executable program can be written in either Fortran or C++.

Number of Executables On a multiple processor machine a cooperating job can be run by starting the same executable on all nodes. All processors run the same code, but the computation proceeds in parallel by each processor working on a different subset of data. This is a **SPMD** model, Single Program Multiple Data.

The alternative is to start a different executable on different processors. This is a **MPMD** model, Multiple Program Multiple Data. There are complications with many job control systems on multiprocessor machines in getting the different executables started, and getting inter-process communcations established. Currently ESMF does not support MPMD.

13.2 Use and Examples

ESMF encourages application organization in which there is a single top-level Gridded Component. This provides a simple, clear sequence of operations at the highest level, and also enables the entire application to be treated as a sub-Gridded Component of another, larger application if desired. When an application is organized in this fashion the standard AppDriver can probably be used without much modification.

Examples of program organization using the AppDriver can be found in the `src/Superstructure/AppDriver` directory. A set of subdirectories within the AppDriver directory follows the naming convention:

`<seq|conc>_<pairwise|hub>_<f|c>driver_<spmd|mpmd>`

The example that is currently implemented is `seq_pairwise_fdriver_spmd`, which has sequential component execution, a pairwise coupler, a main program in Fortran, and all processors launching the same executable.

This simple example is also copied automatically into a top-level `quick_start` directory at compilation time.

The user can copy the AppDriver files into their own local directory. Some of the files can be used unchanged. Others are template files which have the rough outline of the code but need additional application-specific code added in order to perform a meaningful function. The `README` file in the AppDriver subdirectory or quick_start directory contains instructions about which files to change.

```
  !------------------------------------------------------------------------------
  ! The ChangeMe.F90 file contains a number of definitions
  ! that are used by the AppDriver, such as the name of the application's
  ! main configuration file and the name of the application's SetServices
  ! routine.
  !------------------------------------------------------------------------------

 #include "ChangeMe.F90"

     program ESMF_AppDriver

     ! ESMF module, defines all ESMF data types and procedures
     use ESMF_Mod

     ! Gridded Component registration routines.  Defined in "ChangeMe.F90"
     use USER_APP_Mod, only : SetServices => USER_APP_SetServices

     implicit none

     ! Local variables
```

```
   ! Components
   type(ESMF_GridComp) :: compGridded

   ! States, Virtual Machines, and Layouts
   type(ESMF_VM) :: defaultvm
   type(ESMF_DELayout) :: defaultlayout
   type(ESMF_State) :: defaultstate

   ! Configuration information
   type(ESMF_Config) :: config

   ! A common grid
   type(ESMF_Grid) :: grid

   ! A clock, a calendar, and timesteps
   type(ESMF_Clock) :: clock
   type(ESMF_Calendar) :: gregorianCalendar
   type(ESMF_TimeInterval) :: timeStep
   type(ESMF_Time) :: startTime
   type(ESMF_Time) :: stopTime

   ! Variables related to grid and clock
   integer :: i_max, j_max
   real(ESMF_KIND_I8) :: x_min, x_max, y_min, y_max

   ! Return codes for error checks
   integer :: rc
   logical :: dummy

!-------------------------------------------------------------------------------
!  Initialize the ESMF Framework
!-------------------------------------------------------------------------------

   call ESMF_Initialize(rc=rc)
   if (rc .ne. ESMF_SUCCESS) stop 99


   dummy=ESMF_LogWrite("ESMF AppDriver start", ESMF_LOG_INFO)

   !
   ! Read in Configuration information from a default config file
   !

   config = ESMF_ConfigCreate(rc)
   call ESMF_ConfigLoadFile(config, USER_CONFIG_FILE, rc = rc)

   !  *** this section is incomplete. ***
   ! Get standard config parameters, for example:

   ! the default grid size and type
   ! the default start time, stop time, and running intervals
   !  for the main time loop.
   !
   ! e.g. to get an integer parameter from the config file:
   ! call ESMF_ConfigGetAttribute( config, ndays, label ='Number_of_Days:', &
```

```
   !                               default=30, rc = rc )
   !
   call ESMF_ConfigGetAttribute(config, i_max, 'I Counts:', default=20, rc=rc)
   call ESMF_ConfigGetAttribute(config, j_max, 'J Counts:', default=80, rc=rc)
   call ESMF_ConfigGetAttribute(config, x_min, 'X Min:', default=0.0, rc=rc)
   call ESMF_ConfigGetAttribute(config, y_min, 'Y Min:', default=-180.0, rc=rc)
   call ESMF_ConfigGetAttribute(config, x_max, 'X Max:', default=90.0, rc=rc)
   call ESMF_ConfigGetAttribute(config, y_max, 'Y Max:', default=180.0, rc=rc)

!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!    Create section
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------

   ! Get the default VM which contains all PEs this job was started on.
   call ESMF_VMGetGlobal(defaultvm, rc)

   ! Create the top Gridded component, passing in the default layout.
   compGridded = ESMF_GridCompCreate(defaultvm, "ESMF Gridded Component", rc=rc)

   dummy=ESMF_LogWrite("Component Create finished", ESMF_LOG_INFO)


!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!  Register section
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
     call ESMF_GridCompSetServices(compGridded, SetServices, rc)
     if (ESMF_LogMsgFoundError(rc, "Registration failed", rc)) goto 10


!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!  Create and initialize a clock, and a grid.
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------

    ! Based on values from the Config file, create a default Grid
    ! and Clock.  We assume we have read in the variables below from
    ! the config file.


     gregorianCalendar = ESMF_CalendarCreate("Gregorian", &
                                             ESMF_CAL_GREGORIAN, rc)

     call ESMF_TimeIntervalSet(timeStep, S=2, rc=rc)

     call ESMF_TimeSet(startTime, yy=2004, mm=9, dd=25, &
                       calendar=gregorianCalendar, rc=rc)

     call ESMF_TimeSet(stopTime, yy=2004, mm=9, dd=26, &
                       calendar=gregorianCalendar, rc=rc)
```

```
    clock = ESMF_ClockCreate("Application Clock", timeStep, startTime, &
                              stopTime, rc=rc)


    ! Same with the grid.  Get a default layout based on the VM.
    defaultlayout = ESMF_DELayoutCreate(defaultvm, rc=rc)

    grid = ESMF_GridCreateHorzXYUni(counts=(/i_max, j_max/), &
                           minGlobalCoordPerDim=(/x_min, y_min/), &
                           maxGlobalCoordPerDim=(/x_max, y_max/), &
                           horzStagger=ESMF_GRID_HORZ_STAGGER_C_SE, &
                           name="source grid", rc=rc)
    call ESMF_GridDistribute(grid, delayout=defaultlayout, rc=rc)

    ! Attach the Grid to the Component
    call ESMF_GridCompSet(compGridded, grid=grid, rc=rc)


!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!  Create and initialize a State to use for both import and export.
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------

    defaultstate = ESMF_StateCreate("Default Gridded State", rc=rc)

!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!  Init, Run, and Finalize section
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------

    call ESMF_GridCompInitialize(compGridded, defaultstate, defaultstate, &
                                                                clock, rc=rc)
    if (ESMF_LogMsgFoundError(rc, "Initialize failed", rc)) goto 10



    call ESMF_GridCompRun(compGridded, defaultstate, defaultstate, &
                                                                clock, rc=rc)
    if (ESMF_LogMsgFoundError(rc, "Run failed", rc)) goto 10



    call ESMF_GridCompFinalize(compGridded, defaultstate, defaultstate, &
                                                                clock, rc=rc)
    if (ESMF_LogMsgFoundError(rc, "Finalize failed", rc)) goto 10


!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
!     Destroy section
!-------------------------------------------------------------------------------
!-------------------------------------------------------------------------------
```

```
      ! Clean up

      call ESMF_ClockDestroy(clock, rc)

      call ESMF_CalendarDestroy(gregorianCalendar, rc)

      call ESMF_StateDestroy(defaultstate, rc)

      call ESMF_GridCompDestroy(compGridded, rc)

      call ESMF_DELayoutDestroy(defaultLayout, rc)

 !-------------------------------------------------------------------------------
 !-------------------------------------------------------------------------------

10 continue

      call ESMF_Finalize(rc)

      end program ESMF_AppDriver
```

13.3 Restrictions and Future Work

1. **Concurrent components not supported.** Only applications in which components are executed sequentially are supported at this time.

13.4 Required ESMF Methods

13.4.1 ESMF_Initialize - Initialize the ESMF

INTERFACE:

```
      subroutine ESMF_Initialize(defaultConfigFileName, defaultCalendar, &
                                 defaultLogFileName, vm, rc)
```

ARGUMENTS:

```
      character(len=*),          intent(in),  optional :: defaultConfigFileName
      type(ESMF_CalendarType), intent(in),  optional :: defaultCalendar
      character(len=*),          intent(in),  optional :: defaultLogFileName
      type(ESMF_VM),             intent(out), optional :: vm
      integer,                   intent(out), optional :: rc
```

DESCRIPTION:

Initialize the ESMF. This method must be called before any other ESMF methods are used. Before exiting the application the user must call `ESMF_Finalize()` to release resources and clean up the ESMF gracefully.
The arguments are:

[defaultConfigFilename] Name of the default configuration file for the entire application.

[defaultCalendar] Sets the default calendar to be used by ESMF Time Manager. If not specified, defaults to ESMF_CAL_NOCALENDAR

[defaultLogFileName] Name of the default log file for warning and error messages. If not specified, defaults to ESMF_ErrorLog.

[vm] Returns the global ESMF_VM that was created during initialization.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

13.4.2 ESMF_Finalize - Clean up and close the ESMF

INTERFACE:

```
subroutine ESMF_Finalize(rc)
```

ARGUMENTS:

```
integer, intent(out), optional :: rc
```

DESCRIPTION:

Finalize the ESMF. This must be called before the application exits to allow the ESMF to flush buffers, close open connections, and release internal resources cleanly.
The argument is:

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

13.4.3 User-Code SetServices Method

Many programs call some library routines. The library documentation must explain what the routine name is, what arguments are required and what are optional, and what the code does.
In contrast, all ESMF components must be written to *be called* by another part of the program; in effect, an ESMF component takes the place of a library. The interface is prescribed by the framework, and the component writer must provide specific subroutines which have standard argument lists and perform specific operations.
One of the required interfaces a component must provide is the set services method. This subroutine must have an externally accessible name (be a public symbol), take a component as the first argument, and an integer return code as the second. The subroutine name is not predefined, it is set by the component writer, but must be provided as part of the component documentation.
The required function of the set services subroutine is to register the rest of the required functions in the component, currently initialize, run, and finalize methods. The ESMF method ESMF_<Grid/Cpl>CompSetEntryPoint() should be called for each of the required subroutines.
The names of the initialize, run, and finalize user-code subroutines do not need to be public; in fact it is far better for them to be private to lower the chances of public symbol clashes between different components.
Within the set services routine, the user can also register a private data block by calling the ESMF_<Grid|Cpl>CompSetInternalS method.
Note that a component does not call its own set services routine; the AppDriver or parent component code which is creating a component will first call ESMF_<Grid/Cpl>CompCreate() to create an "empty" component, and then call the component-specific set services routine to associate ESMF-standard methods to user-code methods. After set services has been called, the framework now will be able to call the component's initialize, run, and finalize routines as required.

13.4.4 User-Code Initialize, Run, and Finalize Methods

User-code initialize, run, and finalize routines must be provided for each component. See Sections 14.6 and 15.5 for the prescribed interfaces and examples of how to set these up.

14 GridComp Class

14.1 Description

In Earth system modeling, the most natural way to think about an ESMF Gridded Component, or `ESMF_GridComp`, is as a piece of code representing a particular physical domain; for example, an atmospheric model or an ocean model. In many large modeling systems, each components is developed by its own group of domain experts. The ESMF Gridded Component construct provides domain experts with a structured, consistent set of component interfaces so that it is straightforward, at least technically, to combine software from a number of groups, representing different physical domains, to form a complex application.

Earth system software components tend to share a number of basic features. Most contain a variety of physical fields; refer to a (possibly noncontiguous) spatial region and a grid that is partitioned across a set of computational resources; and require a clock, usually for stepping a governing set of PDEs forward in time. Most can also be divided into distinct initialize, run, and finalize computational phases. These common characteristics are used within ESMF to define a Gridded Component data structure that is both tailored for Earth system modeling and yet is still flexible enough to represent a variety of domains.

More broadly, an ESMF Gridded Component can be based on any software with a computational function that is associated with a grid. This might be a convection or radiation scheme, a dynamical core, or a data assimilation system. ESMF allows you to nest Gridded Components, so that the physics and dynamics within an atmospheric model can be considered Gridded Components, along with the atmospheric model itself.

A well-designed Gridded Component does not store information internally about how it couples to other Gridded Components. That allows it to be used in different contexts without changes to source code.[3] Data is passed between Gridded Components using an intermediary Coupler Component, described in Section 15.1.

An ESMF Gridded Component has two parts, one which is user-written and another which is part of the framework. The user-written part is software representing a physical domain or performing some other computational function. It forms the body of the Gridded Component. It may be a piece of legacy code, or it may be developed expressly for use with the ESMF. It must contain routines with standard ESMF interfaces that can be called to initialize, run, and finalize the Gridded Component. These routines can have separate callable phases, such as distinct first and second initialization steps.

The part provided by ESMF is the Gridded Component derived type itself, `ESMF_GridComp`. An `ESMF_GridComp` must be created for every portion of the application that will be represented as a separate component; for example, in a climate model, there may be Gridded Components representing the land, ocean, sea ice, and atmosphere. If the application contains an ensemble of identical Gridded Components, every one has its own associated `ESMF_GridComp`. Each Gridded Component has its own name and is allocated a set of computational resources, in the form of an ESMF Virtual Machine, or VM.

The user-written part of a Gridded Component is associated with an `ESMF_GridComp` derived type through a routine called SetServices. This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call `ESMF_SetEntryPoint` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code.

For example, a user-written initialization routine called `popOceanInit` might be associated with the standard initialize routine of an ESMF Gridded Component named "POP" that represents an ocean model.

14.2 GridComp Options

14.2.1 ESMF_GridCompType

DESCRIPTION:

The `ESMF_GridCompType` flag identifies what sort of physical domain or computational function a particular `ESMF_GridComp` represents. The flag values are purely informational; they are not used anywhere within the framework. Use of this flag is optional.

Valid values are:

ESMF_ATM Atmospheric model.

[3] The idea here is to avoid situations in which slightly different versions of the same model source are maintained for use in different contexts - standalone vs. coupled versions, for example.

ESMF_LAND Land model.

ESMF_OCEAN Ocean model.

ESMF_SEAICE Sea ice model.

ESMF_RIVER River model.

ESMF_OTHER Other type of model or system.

14.3 Use and Examples

A Gridded Component is a computational entity which consumes and produces data. It uses a State object to exchange data between itself and other Components. It uses a Clock object to manage time, and a VM to describe its own and its child components' computational resources.

```
! !PROGRAM: ESMF_GCompEx.F90 - Gridded Component example
!
! !DESCRIPTION:
!
!   The skeleton of one of many possible Gridded component models.
!
!-----------------------------------------------------------------------------

    ! Example Gridded Component
    module ESMF_GriddedCompEx

    ! ESMF Framework module
    use ESMF_Mod
    implicit none
    public GComp_SetServices

    contains
```

14.3.1 Specifying a User-Code SetServices Routine

Every `ESMF_GridComp` is required to provide and document a set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_GridComp` as the first argument, and an integer return code as the second.
It must call the ESMF method `ESMF_GridCompSetEntryPoint()` to register with the framework what user-code subroutines should be called to initialize, run, and finalize the component. There are additional routines which can be registered as well, for checkpoint and restart functions.
Note that the actual subroutines being registered do not have to be public to this module; only the set services routine itself must be available to be used by other code.

```
    subroutine GComp_SetServices(comp, rc)
        type(ESMF_GridComp) :: comp
        integer :: rc

        ! SetServices the callback routines.
        call ESMF_GridCompSetEntryPoint(comp, ESMF_SETINIT, GComp_Init, 0, rc)
        call ESMF_GridCompSetEntryPoint(comp, ESMF_SETRUN, GComp_Run, 0, rc)
        call ESMF_GridCompSetEntryPoint(comp, ESMF_SETFINAL, GComp_Final, 0, rc)
```

```
        ! If desired, this routine can register a private data block
        ! to be passed in to the routines above:
        ! call ESMF_GridCompSetData(comp, mydatablock, rc)

        rc = ESMF_SUCCESS

    end subroutine
```

14.3.2 Specifying a User-Code Initialize Routine

When a higher level component is ready to begin using an `ESMF_GridComp`, it will call its initialize routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.
At initialization time the component can allocate data space, open data files, set up initial conditions; anything it needs to do to prepare to run.
The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
    subroutine GComp_Init(comp, importState, exportState, clock, rc)
        type(ESMF_GridComp) :: comp
        type(ESMF_State) :: importState, exportState
        type(ESMF_Clock) :: clock
        integer :: rc

        print *, "Gridded Comp Init starting"

        ! This is where the model specific setup code goes.

        ! If the initial Export state needs to be filled, do it here.
        !call ESMF_StateAddField(exportState, field, rc)
        !call ESMF_StateAddBundle(exportState, bundle, rc)
        print *, "Gridded Comp Init returning"

        rc = ESMF_SUCCESS

    end subroutine GComp_Init
```

14.3.3 Specifying a User-Code Run Routine

During the execution loop, the run routine may be called many times. Each time it should read data from the `importState`, use the `clock` to determine what the current time is in the calling component, compute new values or process the data, and produce any output and place it in the `exportState`.
When a higher level component is ready to use the `ESMF_GridComp` it will call its run routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.
It is expected that this is where the bulk of the model computation or data analysis will occur.
The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
    subroutine GComp_Run(comp, importState, exportState, clock, rc)
        type(ESMF_GridComp) :: comp
        type(ESMF_State) :: importState, exportState
```

```
    type(ESMF_Clock) :: clock
    integer :: rc

    print *, "Gridded Comp Run starting"
    ! call ESMF_StateGetField(), etc to get fields, bundles, arrays
    !  from import state.

    ! This is where the model specific computation goes.

    ! Fill export state here using ESMF_StateAddField(), etc

    print *, "Gridded Comp Run returning"

    rc = ESMF_SUCCESS

end subroutine GComp_Run
```

14.3.4 Specifying a User-Code Finalize Routine

At the end of application execution, each `ESMF_GridComp` should deallocate data space, close open files, and flush final results. These functions should be placed in a finalize routine.
The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
subroutine GComp_Final(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp) :: comp
    type(ESMF_State) :: importState, exportState
    type(ESMF_Clock) :: clock
    integer :: rc

    print *, "Gridded Comp Final starting"

    ! Add whatever code here needed

    print *, "Gridded Comp Final returning"

    rc = ESMF_SUCCESS

end subroutine GComp_Final

end module ESMF_GriddedCompEx
```

14.4 Restrictions and Future Work

1. **No concurrent components.** While the design of concurrently running Components is fairly complete, this release of the framework does not support them. Only sequentially executing Components can be run.

2. **No Transforms.** Components must exchange data through `ESMF_State` objects. The input data are available at the time the user Component code is called, and data to be returned to another Component are available when that code returns. `ESMF_Xform` objects provide a way for a Component to prepare data to be transformed and sent to another Component from within the execution of the user Component code. Transforms are not implemented in this version of the framework.

3. **Data isolation.** Gridded Components must only communicate with other components via data in State objects. They must not make direct references to data in other States.

4. **Namespace isolation.** If possible, Gridded Components should attempt to make all data private, so public names do not interfere with data in other components.

5. **Single execution mode.** It is not expected that a single Gridded Component be able to function in both sequential and concurrent modes, although Gridded Components of different types can be nested. For example, a concurrently called Gridded Component can contain several nested sequential Gridded Components. However, only sequentially executing Components are supported in this release of the framework.

14.5 Class API: Basic GridComp Methods

14.5.1 ESMF_GridCompCreate - Create a new GridComp from a Config file

INTERFACE:

```
! Private name; call using ESMF_GridCompCreate()
function ESMF_GridCompCreateConf(name, gridcomptype, grid, &
                                 config, configFile, clock, rc)
```

RETURN VALUE:

```
type(ESMF_GridComp) :: ESMF_GridCompCreateConf
```

ARGUMENTS:

```
!external :: services
character(len=*), intent(in), optional :: name
type(ESMF_GridCompType), intent(in), optional :: gridcomptype
type(ESMF_Grid), intent(in), optional :: grid
type(ESMF_Config), intent(in), optional :: config
character(len=*), intent(in), optional :: configFile
type(ESMF_Clock), intent(inout), optional :: clock
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a new `ESMF_GridComp`, specifying optional configuration file and other information.
The return value is the new `ESMF_GridComp`.
The arguments are:

[name] Name of the newly-created `ESMF_GridComp`. This name can be altered from within the `ESMF_GridComp` code once the initialization routine is called.

[gridcomptype] `ESMF_GridComp` model type, where model includes `ESMF_ATM, ESMF_LAND, ESMF_OCEAN, ESMF_SEAICE, ESMF_RIVER`. Note that this has no meaning to the framework, it is an annotation for user code to query.

[grid] Default `ESMF_Grid` associated with this `gridcomp`.

[config] An already-created `ESMF_Config` configuration object from which the new `ESMF_GridComp` can read in namelist-type information to set parameters for this run.

[configFile] The filename of an `ESMF_Config` format file. If specified, this file is opened an `ESMF_Config` configuration object is created for the file and attached to the new `ESMF_GridComp`. The user can call `ESMF_GridCompGet()` to get and use the object. If both are specified, the `config` object takes priority over this one.

[clock] Component-specific ESMF_Clock. This clock is available to be queried and updated by the new ESMF_GridComp as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.2 ESMF_GridCompCreate - Create a new GridComp with VM enabled

INTERFACE:

```
! Private name; call using ESMF_GridCompCreate()
function ESMF_GridCompCreateVM(vm, name, gridcomptype, grid, &
                               config, configFile, clock, petList, rc)
```

RETURN VALUE:

```
type(ESMF_GridComp) :: ESMF_GridCompCreateVM
```

ARGUMENTS:

```
!external :: services
type(ESMF_VM),             intent(in)                :: vm
character(len=*),          intent(in),    optional :: name
type(ESMF_GridCompType), intent(in),    optional :: gridcomptype
type(ESMF_Grid),           intent(in),    optional :: grid
type(ESMF_Config),         intent(in),    optional :: config
character(len=*),          intent(in),    optional :: configFile
type(ESMF_Clock),          intent(inout), optional :: clock
integer,                   intent(in),    optional :: petList(:)
integer,                   intent(out),   optional :: rc
```

DESCRIPTION:

Create a new ESMF_GridComp, setting the resources explicitly.
The return value is the new ESMF_GridComp.
The arguments are:

vm ESMF_VM object for the parent component out of which this ESMF_GridCompCreate() call is issued. This will become the parent ESMF_VM of the newly create ESMF_GridComp.

[name] Name of the newly-created ESMF_GridComp. This name can be altered from within the ESMF_GridComp code once the initialization routine is called.

[gridcomptype] ESMF_GridComp model type, where model includes ESMF_ATM, ESMF_LAND, ESMF_OCEAN, ESMF_SEAICE, ESMF_RIVER. Note that this has no meaning to the framework, it is an annotation for user code to query.

[grid] Default ESMF_Grid associated with this gridcomp.

[config] An already-created ESMF_Config configuration object from which the new component can read in namelist-type information to set parameters for this run. If both are specified, this object takes priority over configFile.

[configFile] The filename of an ESMF_Config format file. If specified, this file is opened an ESMF_Config configuration object is created for the file, and attached to the new component. The user can call ESMF_GridCompGet() to get and use the object. If both are specified, the config object takes priority over this one.

[clock] Component-specific ESMF_Clock. This clock is available to be queried and updated by the new ESMF_GridComp as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[petList] List of PETs in the given ESMF_VM that the parent component is giving to the created child component. If petList is not specified all of the parents PETs will be given to the child component.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.3 ESMF_GridCompDestroy - Release resources for a GridComp

INTERFACE:

```
subroutine ESMF_GridCompDestroy(gridcomp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this ESMF_GridComp.
The arguments are:

gridcomp Release all resources associated with this ESMF_GridComp and mark the object as invalid. It is an error to pass this object into any other routines after being destroyed.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.4 ESMF_GridCompFinalize - Call the GridComp's finalize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompFinalize(gridcomp, importState, &
                             exportState, clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                                  :: gridcomp
type (ESMF_State),        intent(inout), optional :: importState
type (ESMF_State),        intent(inout), optional :: exportState
type (ESMF_Clock),        intent(in),    optional :: clock
integer,                  intent(in),    optional :: phase
type (ESMF_BlockingFlag), intent(in),    optional :: blockingflag
integer,                  intent(out),   optional :: rc
```

DESCRIPTION:

Call the associated user-supplised finalization code for an ESMF_GridComp.
The arguments are:

gridcomp The ESMF_GridComp to call finalize routine for.

[importState] ESMF_State containing import data.

[exportState] ESMF_State containing export data.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be ESMF_SINGLEPHASE. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Use ESMF_BLOCKING (default) or ESMF_NONBLOCKING.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.5 ESMF_GridCompGet - Query a GridComp for information

INTERFACE:

```
subroutine ESMF_GridCompGet(gridcomp, name, gridcomptype, &
                            grid, config, configFile, clock, vm, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),      intent(in)            :: gridcomp
character(len=*),         intent(out), optional :: name
type(ESMF_GridCompType), intent(out), optional :: gridcomptype
type(ESMF_Grid),          intent(out), optional :: grid
type(ESMF_Config),        intent(out), optional :: config
character(len=*),         intent(out), optional :: configFile
type(ESMF_Clock),         intent(out), optional :: clock
type(ESMF_VM),            intent(out), optional :: vm
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Returns information about an ESMF_GridComp. For queries where the caller only wants a single value, specify the argument by name. All the arguments after the gridcomp argument are optional to facilitate this.
The arguments are:

gridcomp ESMF_GridComp object to query.

[name] Return the name of the ESMF_GridComp.

[gridcomptype] Return the model type of this ESMF_GridComp.

[grid] Return the ESMF_Grid associated with this ESMF_GridComp.

[config] Return the ESMF_Config object for this ESMF_GridComp.

[configFile] Return the configuration filename for this ESMF_GridComp.

[clock] Return the private clock for this ESMF_GridComp.

[vm] Return the ESMF_VM for this ESMF_GridComp.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.6 ESMF_GridCompInitialize - Call the GridComp's initialize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompInitialize(gridcomp, importState, &
                            exportState, clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                                 :: gridcomp
type (ESMF_State),        intent(inout), optional :: importState
type (ESMF_State),        intent(inout), optional :: exportState
type (ESMF_Clock),        intent(in),    optional :: clock
integer,                  intent(in),    optional :: phase
type (ESMF_BlockingFlag), intent(in),    optional :: blockingflag
integer,                  intent(out),   optional :: rc
```

DESCRIPTION:

Call the associated user initialization code for a gridcomp.
The arguments are:

gridcomp ESMF_GridComp to call initialize routine for.

[importState] ESMF_State containing import data for coupling.

[exportState] ESMF_State containing export data for coupling.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be ESMF_SINGLEPHASE. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Valid values are ESMF_BLOCKING (the default) or ESMF_NONBLOCKING.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.7 ESMF_GridCompPrint - Print the contents of a GridComp

INTERFACE:

```
subroutine ESMF_GridCompPrint(gridcomp, options, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about an ESMF_GridComp to stdout.
The arguments are:

gridcomp ESMF_GridComp to print.

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.8 ESMF_GridCompReadRestart - Call the GridComp's restore routine

INTERFACE:

```
recursive subroutine ESMF_GridCompReadRestart(gridcomp, iospec, clock, &
                                               phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp), intent(inout) :: gridcomp
type (ESMF_IOSpec), intent(inout), optional :: iospec
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user restore code for a gridcomp.
The arguments are:

gridcomp ESMF_GridComp object to call readrestart routine for.

[iospec] ESMF_IOSpec object which describes I/O options.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. ESMF_State containing export data for coupling.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be ESMF_SINGLEPHASE. For multiple-phase child components, this is the integer phase number to be invoked. If multiple-phase restore, which phase number this is. Pass in 0 or ESMF_SINGLEPHASE for non-multiples.

[blockingflag] Valid values are ESMF_BLOCKING (the default) or ESMF_NONBLOCKING.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.5.9 ESMF_GridCompRun - Call the GridComp's run routine

INTERFACE:

```
recursive subroutine ESMF_GridCompRun(gridcomp, importState, exportState,&
                                      clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                                :: gridcomp
type (ESMF_State),        intent(inout), optional :: importState
type (ESMF_State),        intent(inout), optional :: exportState
type (ESMF_Clock),        intent(in),    optional :: clock
integer,                  intent(in),    optional :: phase
type (ESMF_BlockingFlag), intent(in),    optional :: blockingflag
integer,                  intent(out),   optional :: rc
```

DESCRIPTION:

Call the associated user run code for an `ESMF_GridComp`.
The arguments are:

gridcomp `ESMF_GridComp` to call run routine for.

[importState] `ESMF_State` containing import data.

[exportState] `ESMF_State` containing export data.

[clock] External clock for passing in time information.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked. If multiple-phase restore, which phase number this is. Pass in 0 or `ESMF_SINGLEPHASE` for non-multiples.

[blockingflag] Use `ESMF_BLOCKING` (default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.5.10 ESMF_GridCompSet - Set or reset information about the GridComp

INTERFACE:

```
subroutine ESMF_GridCompSet(gridcomp, name, gridcomptype, grid, &
                            config, configFile, clock, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),     intent(inout)          :: gridcomp
character(len=*),        intent(in),  optional :: name
type(ESMF_GridCompType), intent(in),  optional :: gridcomptype
type(ESMF_Grid),         intent(in),  optional :: grid
type(ESMF_Config),       intent(in),  optional :: config
character(len=*),        intent(in),  optional :: configFile
type(ESMF_Clock),        intent(in),  optional :: clock
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets or resets information about an `ESMF_GridComp`. The caller can set individual values by specifying the arguments by name. All the arguments except `gridcomp` are optional to facilitate this.
The arguments are:

gridcomp `ESMF_GridComp` to change.

[name] Set the name of the `ESMF_GridComp`.

[gridcomptype] Set the model type for this `ESMF_GridComp`.

[grid] Set the `ESMF_Grid` associated with the `ESMF_GridComp`.

[config] Set the configuration information for the `ESMF_GridComp` from this already created `ESMF_Config` object. If specified, takes priority over `configFile`.

[configFile] Set the configuration filename for this `ESMF_GridComp`. An `ESMF_Config` object will be created for this file and attached to the `ESMF_GridComp`. Superceeded by `config` if both are specified.

[clock] Set the private clock for this `ESMF_GridComp`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.5.11 ESMF_GridCompValidate - Check validity of a GridComp

INTERFACE:

```
subroutine ESMF_GridCompValidate(gridcomp, options, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Currently all this method does is to check that the `gridcomp` exists.
The arguments are:

gridcomp `ESMF_GridComp` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.5.12 ESMF_GridCompWriteRestart - Call the GridComp's checkpoint routine

INTERFACE:

```
recursive subroutine ESMF_GridCompWriteRestart(gridcomp, iospec, clock, &
                                               phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp), intent(inout) :: gridcomp
type (ESMF_IOSpec), intent(inout), optional :: iospec
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user checkpoint code for an `ESMF_GridComp`.
The arguments are:

gridcomp `ESMF_GridComp` to call writerestart routine for.

[iospec] `ESMF_IOSpec` object which describes I/O options.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Valid values are `ESMF_BLOCKING` (the default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.5.13 ESMF_GridCompWait - Wait for a GridComp to return

INTERFACE:

```
subroutine ESMF_GridCompWait(gridcomp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)             :: gridcomp
integer,             intent(out), optional  :: rc
```

DESCRIPTION:

When executing asychronously, wait for an `ESMF_GridComp` to return.
The arguments are:

gridcomp `ESMF_GridComp` to wait for.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.6 Class API: SetServices and Related Methods

14.6.1 ESMF_GridCompGetInternalState - Get private data block pointer

INTERFACE:

```
      subroutine ESMF_GridCompGetInternalState(gridcomp, dataPointer, rc)
```

ARGUMENTS:

```
      type(ESMF_GridComp), intent(inout) :: gridcomp
      type(any), pointer, intent(in) :: dataPointer
      integer, intent(out), optional :: rc
```

DESCRIPTION:

Available to be called by an `ESMF_GridComp` at any time after `ESMF_GridCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block, and the address of that block can be registered with the framework and retrieved by this call. When running multiple instantiations of an `ESMF_GridComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMF_GridCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer.
The arguments are:

gridcomp An `ESMF_GridComp` object.

dataPointer A derived type, containing only a pointer to the private data block. The framework will fill in the block and when this call returns the pointer is set to the same address set during `ESMF_GridCompSetInternalState`. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

14.6.2 ESMF_GridCompSetEntryPoint - Set name of GridComp subroutines

INTERFACE:

```
      subroutine ESMF_GridCompSetEntryPoint(gridcomp, subroutineType, &
                                            subroutineName, phase, rc)
```

ARGUMENTS:

```
      type(ESMF_GridComp), intent(inout) :: gridcomp
      character(len=*), intent(in) :: subroutineType
      subroutine, intent(in) :: subroutineName
      integer, intent(in) :: phase
      integer, intent(out), optional :: rc
```

DESCRIPTION:

Intended to be called by an `ESMF_GridComp` during the registration process. An `ESMF_GridComp` calls `ESMF_GridCompSetEnt` for each of the predefined init, run, and finalize routines, to assocate the internal subroutine to be called for each function. If multiple phases for init, run, or finalize are needed, this can be called with phase numbers.
After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for this child `ESMF_GridComp`.
The arguments are:

gridcomp An ESMF_GridComp object.

subroutineType One of a set of predefined subroutine types - e.g. ESMF_SETINIT, ESMF_SETRUN, ESMF_SETFINAL.

subroutineName The name of the gridcomp subroutine to be associated with the subroutineType. This subroutine does not have to be public to the module.

[phase] For ESMF_GridComps which need to initialize or run or finalize with mutiple phases, the phase number which corresponds to this subroutine name. For single phase subroutines, either omit this argument, or use the parameter ESMF_SINGLEPHASE. The ESMF_GridComp writer must document the requirements of the ESMF_GridComp for how and when the multiple phases are expected to be called.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.6.3 ESMF_GridCompSetInternalState - Set private data block pointer

INTERFACE:

```
subroutine ESMF_GridCompSetInternalState(gridcomp, dataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(inout) :: gridcomp
type(any), pointer, intent(in) :: dataPointer
integer, intent(out), optional :: rc
```

DESCRIPTION:

Available to be called by an ESMF_GridComp at any time, but expected to be most useful when called during the registration process, or initialization. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block, and the address of that block can be registered with the framework and retrieved by subsequent calls. When running multiple instantiations of an ESMF_GridComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMF_GridCompGetInternalState call retrieves the data pointer.
The arguments are:

gridcomp An ESMF_GridComp object.

dataPointer A pointer to the private data block, wrapped in a derived type which contains only a pointer to the block. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

14.6.4 ESMF_GridCompSetServices - Register GridComp interface routines

INTERFACE:

```
subroutine ESMF_GridCompSetServices(gridcomp, subroutineName, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp
subroutine :: subroutineName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call a gridded ESMF_GridComp's setservices registration routine. The parent component must first create an ESMF_GridComp, then call this routine. The arguments are the object returned from the create call, plus the public, well-known subroutine name that is the registration routine for this ESMF_GridComp. This name must be documented by the ESMF_GridComp provider.
After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for the ESMF_GridComp.
The arguments are:

gridcomp An ESMF_GridComp object.

subroutineName The public name of the gridcomp's ESMF_GridCompSetServices call. An ESMF_GridComp writer must provide this information. Note that this is the actual subroutine, not a character string.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15 CplComp Class

15.1 Description

In a large, multi-component application such as a weather forecasting or climate prediction system running within ESMF, physical domains and major system functions are represented as Gridded Components (see Section **??**). A Coupler Component, or ESMF_CplComp, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and in another coupled to a data assimilation system for numerical weather prediction.
Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange between Gridded Components. The term "user-written" is somewhat misleading here, since within a Coupler Component the user can leverage ESMF infrastructure software for regridding, redistribution, lower-level communications, calendar management, and other functions. However, ESMF is unlikely to offer all the software necessary to customize a data transfer between Gridded Components. ESMF does not currently offer tools for unit tranformations or time averaging operations, so users must manage those operations themseves.
The user-written Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.
The second part of a Coupler Component is the ESMF_CplComp derived type within ESMF. The user must create one of these types to represent a specific coupling function, such as the regular transfer of data between a data assimilation system and an atmospheric model. [4]
The user-written part of a Coupler Component is associated with an ESMF_CplComp derived type through a routine called SetServices. This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call ESMF_SetEntryPoint methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code. For example, a user routine called "couplerInit" might be associated with the standard initialize routine in a Coupler Component.
Coupler Components can be written to transform data between a pair of Gridded Components, or a single Coupler Component can couple more than two Gridded Components.

[4] It is not necessary to create a Coupler Component for each individual data *transfer.*

15.2 Use and Examples

A Coupler Component manages the transformation of data between Components. It contains a list of State objects and the operations needed to make them compatible, including such things as regridding and unit conversion. Coupler Components are user-written, following prescribed ESMF interfaces and, wherever desired, using ESMF infrastructure tools.

```
! !PROGRAM: ESMF_CplEx.F90 - Coupler Component example
!
! !DESCRIPTION:
!
!   The skeleton of one of many possible Coupler component models.
!
!-------------------------------------------------------------------------------

    ! Example Coupler Component
    module ESMF_CouplerEx

    ! ESMF Framework module
    use ESMF_Mod
    implicit none
    public CPL_SetServices

    contains
```

15.2.1 Specifying a User-Code SetServices Routine

Every `ESMF_CplComp` is required to provide and document a set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_CplComp` as the first argument, and an integer return code as the second.
It must call the ESMF method `ESMF_CplCompSetEntryPoint()` to register with the framework what user-code subroutines should be called to initialize, run, and finalize the component. There are additional routines which can be registered as well, for checkpoint and restart functions.
Note that the actual subroutines being registered do not have to be public to this module; only the set services routine itself must be available to be used by other code.

```
    subroutine CPL_SetServices(comp, rc)
        type(ESMF_CplComp) :: comp
        integer :: rc

        ! SetServices the callback routines.
        call ESMF_CplCompSetEntryPoint(comp, ESMF_SETINIT, CPL_Init, 0, rc)
        call ESMF_CplCompSetEntryPoint(comp, ESMF_SETRUN, CPL_Run, 0, rc)
        call ESMF_CplCompSetEntryPoint(comp, ESMF_SETFINAL, CPL_Final, 0, rc)

        ! If desired, this routine can register a private data block
        ! to be passed in to the routines above:
        ! call ESMF_CplCompSetInternalState(comp, mydatablock, rc)

        rc = ESMF_SUCCESS
    end subroutine
```

15.2.2 Specifying a User-Code Initialize Routine

When a higher level component is ready to begin using an ESMF_CplComp, it will call its initialize routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.
At initialization time the component can allocate data space, open data files, set up initial conditions; anything it needs to do to prepare to run.
The rc return code should be set if an error occurs, otherwise the value ESMF_SUCCESS should be returned.

```
subroutine CPL_Init(comp, importState, exportState, clock, rc)
    type(ESMF_CplComp) :: comp
    type(ESMF_State) :: importState
    type(ESMF_State) :: exportState
    type(ESMF_Clock) :: clock
    integer :: rc
    type(ESMF_State) :: nestedstate

    print *, "Coupler Init starting"

    ! Add whatever code here needed
    ! Precompute any needed values, fill in any inital values
    !  needed in Import States

    rc = ESMF_SUCCESS

    print *, "Coupler Init returning"

end subroutine CPL_Init
```

15.2.3 Specifying a User-Code Run Routine

During the execution loop, the run routine may be called many times. Each time it should read data from the importState, use the clock to determine what the current time is in the calling component, compute new values or process the data, and produce any output and place it in the exportState.
When a higher level component is ready to use the ESMF_CplComp it will call its run routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.
It is expected that this is where the bulk of the model computation or data analysis will occur.
The rc return code should be set if an error occurs, otherwise the value ESMF_SUCCESS should be returned.

```
subroutine CPL_Run(comp, importState, exportState, clock, rc)
    type(ESMF_CplComp) :: comp
    type(ESMF_State) :: importState
    type(ESMF_State) :: exportState
    type(ESMF_Clock) :: clock
    integer :: rc

    type(ESMF_State) :: nestedstate

    print *, "Coupler Run starting"

    ! Add whatever code needed here to transform Export state data
    !  into Import states for the next timestep.
```

```
        rc = ESMF_SUCCESS

        print *, "Coupler Run returning"

    end subroutine CPL_Run
```

15.2.4 Specifying a User-Code Finalize Routine

At the end of application execution, each ESMF_CplComp should deallocate data space, close open files, and flush final results. These functions should be placed in a finalize routine.
The rc return code should be set if an error occurs, otherwise the value ESMF_SUCCESS should be returned.

```
    subroutine CPL_Final(comp, importState, exportState, clock, rc)
        type(ESMF_CplComp) :: comp
        type(ESMF_State) :: importState
        type(ESMF_State) :: exportState
        type(ESMF_Clock) :: clock
        integer :: rc

        type(ESMF_State) :: nestedstate

        print *, "Coupler Final starting"

        ! Add whatever code needed here to compute final values and
        !  finish the computation.

        rc = ESMF_SUCCESS

        print *, "Coupler Final returning"

    end subroutine CPL_Final

    end module ESMF_CouplerEx
```

15.3 Restrictions and Future Work

1. **No concurrent components.** While the design of concurrently running components is fairly complete, this release of the framework does not support them. Only sequentially executing components can be run.
2. **No Transforms.** Components must exchange data through ESMF_State objects. The input data are available at the time the component code is called, and data to be returned to another component are available when that code returns. ESMF_Xform objects provide a way for a component to prepare data to be transformed and sent to another component from within the execution of the component code. Transforms are not implemented in this version of the framework.
3. **No automatic unit conversions.** The ESMF framework does not currently contain tools for performing unit conversions, operations that are fairly standard within Coupler Components.
4. **No accumulator.** The ESMF does not have an accumulator tool, to perform time averaging of fields for coupling. This is likely to be developed in the near term.

15.4 Class API: Basic CplComp Methods

15.4.1 ESMF_CplCompCreate - Create a new CplComp from a Config file

INTERFACE:

```
! Private name; call using ESMF_CplCompCreate()
function ESMF_CplCompCreateConf(name, config, configFile, clock, rc)
```

RETURN VALUE:

```
type(ESMF_CplComp) :: ESMF_CplCompCreateConf
```

ARGUMENTS:

```
character(len=*), intent(in), optional :: name
type(ESMF_Config), intent(in), optional :: config
character(len=*), intent(in), optional :: configFile
type(ESMF_Clock), intent(in), optional :: clock
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a new `ESMF_CplComp`, specifying optional configuration file information.
The return value is the new `ESMF_CplComp`.
The arguments are:

[name] Name of the newly-created `ESMF_CplComp`. This name can be altered from within the `ESMF_CplComp` code once the initialization routine is called.

[config] An already-created `ESMF_Config` configuration object from which the new `ESMF_CplComp` can read in namelist-type information to set parameters for this run. If both are specified, this object takes priority over `configFile`.

[configFile] The filename of an `ESMF_Config` format file. If specified, this file is opened, an `ESMF_Config` configuration object is created for the file and attached to the new `ESMF_CplComp`. The user can call `ESMF_CplCompGet()` to get and use the object. If both are specified, the `config` object takes priority over this one.

[clock] Component-specific `ESMF_Clock`. This clock is available to be queried and updated by the new `ESMF_CplComp` as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.2 ESMF_CplCompCreate - Create a new CplComp with VM enabled

INTERFACE:

```
! Private name; call using ESMF_CplCompCreate()
function ESMF_CplCompCreateVM(vm, name, config, configFile, &
                              clock, petList, rc)
```

RETURN VALUE:

```
type(ESMF_CplComp) :: ESMF_CplCompCreateVM
```

ARGUMENTS:

```
type(ESMF_VM),      intent(in)            :: vm
character(len=*),   intent(in),  optional :: name
type(ESMF_Config),  intent(in),  optional :: config
character(len=*),   intent(in),  optional :: configFile
type(ESMF_Clock),   intent(in),  optional :: clock
integer,            intent(in),  optional :: petList(:)
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Create a new ESMF_CplComp, setting the resources explicitly.
The return value is the new ESMF_CplComp.
The arguments are:

vm ESMF_VM object for the parent component out of which this ESMF_CplCompCreate() call is issued. This will become the parent ESMF_VM of the newly created ESMF_CplComp.

[name] Name of the newly-created ESMF_CplComp. This name can be altered from within the ESMF_CplComp code once the initialization routine is called.

[config] An already-created ESMF_Config configuration object from which the new component can read in namelist-type information to set parameters for this run. If both are specified, this object takes priority over configFile.

[configFile] The filename of an ESMF_Config format file. If specified, this file is opened, an ESMF_Config configuration object is created for the file, and attached to the new component. The user can call ESMF_CplCompGet() to get and use the object. If both are specified, the config object takes priority over this one.

[clock] Component-specific ESMF_Clock. This clock is available to be queried and updated by the new ESMF_CplComp as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[petList] List of PETs in the given ESMF_VM that the parent component is giving to the created child component. If petList is not specified all of the parents PETs will be given to the child component.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.4.3 ESMF_CplCompDestroy - Release resources for a CplComp

INTERFACE:

```
subroutine ESMF_CplCompDestroy(cplcomp, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this ESMF_CplComp.
The arguments are:

cplcomp Release all resources associated with this ESMF_CplComp and mark the object as invalid. It is an error to pass this object into any other routines after being destroyed.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.4.4 ESMF_CplCompFinalize - Call the CplComp's finalize routine

INTERFACE:

```
     recursive subroutine ESMF_CplCompFinalize(cplcomp, importState, &
                                  exportState, clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
       type (ESMF_CplComp) :: cplcomp
       type (ESMF_State), intent(inout), optional :: importState
       type (ESMF_State), intent(inout), optional :: exportState
       type (ESMF_Clock), intent(in), optional :: clock
       integer, intent(in), optional :: phase
       type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user-supplied finalization routine for an `ESMF_CplComp`.
The arguments are:

cplcomp The `ESMF_CplComp` to call finalize routine for.

[importState] `ESMF_State` containing import data for coupling.

[exportState] `ESMF_State` containing export data for coupling.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Use `ESMF_BLOCKING` (default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.5 ESMF_CplCompGet - Query a CplComp for information

INTERFACE:

```
       subroutine ESMF_CplCompGet(cplcomp, name, config, &
                                                 configFile, clock, vm, rc)
```

ARGUMENTS:

```
       type(ESMF_CplComp), intent(in) :: cplcomp
       character(len=*), intent(out), optional :: name
       type(ESMF_Config), intent(out), optional :: config
       character(len=*), intent(out), optional :: configFile
       type(ESMF_Clock), intent(out), optional :: clock
       type(ESMF_VM), intent(out), optional :: vm
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information about an `ESMF_CplComp`. For queries where the caller only wants a single value, specify the argument by name. All the arguments after `cplcomp` argument are optional to facilitate this.
The arguments are:

cplcomp `ESMF_CplComp` to query.

[name] Return the name of the `ESMF_CplComp`.

[config] Return the `ESMF_Config` object for this `ESMF_CplComp`.

[configFile] Return the configuration filename for this `ESMF_CplComp`.

[clock] Return the private clock for this `ESMF_CplComp`.

[vm] Return the `ESMF_VM` for this `ESMF_CplComp`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.6 ESMF_CplCompInitialize - Call the CplComp's initialize routine

INTERFACE:

```
recursive subroutine ESMF_CplCompInitialize(cplcomp, importState, &
                              exportState, clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp) :: cplcomp
type (ESMF_State), intent(inout), optional :: importState
type (ESMF_State), intent(inout), optional :: exportState
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user initialization code for an `ESMF_CplComp`.
The arguments are:

cplcomp `ESMF_CplComp` to call initialize routine for.

[importState] `ESMF_State` containing import data for coupling.

[exportState] `ESMF_State` containing export data for coupling.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Valid values are `ESMF_BLOCKING` (the default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.7 ESMF_CplCompPrint - Print the contents of a CplComp

INTERFACE:

```
subroutine ESMF_CplCompPrint(cplcomp, options, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about an `ESMF_CplComp` to `stdout`.
The arguments are:

cplcomp `ESMF_CplComp` to print.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.8 ESMF_CplCompReadRestart – Call the CplComp's restore routine

INTERFACE:

```
recursive subroutine ESMF_CplCompReadRestart(cplcomp, iospec, clock, &
                                             phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp), intent(inout) :: cplcomp
type (ESMF_IOSpec), intent(inout), optional :: iospec
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user restore code for an `ESMF_CplComp`.
The arguments are:

cplcomp `ESMF_CplComp` to call readrestart routine for.

[iospec] `ESMF_IOSpec` object which describes I/O options.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. `ESMF_State` containing export data for coupling.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked. If multiple-phase restore, which phase number this is. Pass in 0 or `ESMF_SINGLEPHASE` for non-multiples.

[blockingflag] Valid values are `ESMF_BLOCKING` (the default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.9 ESMF_CplCompRun - Call the CplComp's run routine

INTERFACE:

```
recursive subroutine ESMF_CplCompRun(cplcomp, importState, exportState, &
                                     clock, phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp) :: cplcomp
type (ESMF_State), intent(inout), optional :: importState
type (ESMF_State), intent(inout), optional :: exportState
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user run code for an `ESMF_CplComp`.
The arguments are:

cplcomp `ESMF_CplComp` to call run routine for.

[importState] `ESMF_State` containing import data for coupling.

[exportState] `ESMF_State` containing export data for coupling.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be `ESMF_SINGLEPHASE`. For multiple-phase child components, this is the integer phase number to be invoked. If multiple-phase restore, which phase number this is. Pass in 0 or `ESMF_SINGLEPHASE` for non-multiples. External clock for passing in time information.

[blockingflag] Use `ESMF_BLOCKING` (default) or `ESMF_NONBLOCKING`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.4.10 ESMF_CplCompSet - Set or reset information about the CplComp

INTERFACE:

```
subroutine ESMF_CplCompSet(cplcomp, name, config, configFile, clock, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(inout) :: cplcomp
character(len=*), intent(in), optional :: name
type(ESMF_Config), intent(in), optional :: config
character(len=*), intent(in), optional :: configFile
type(ESMF_Clock), intent(in), optional :: clock
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets or resets information about an ESMF_CplComp. The caller can set individual values by specifying the arguments by name. All the arguments except cplcomp are optional to facilitate this.
The arguments are:

cplcomp ESMF_CplComp to change.

[name] Set the name of the ESMF_CplComp.

[config] Set the configuration information for the ESMF_CplComp from this already created ESMF_Config object. If specified, takes priority over configFile.

[configFile] Set the configuration filename for this ESMF_CplComp. An ESMF_Config object will be created for this file and attached to the ESMF_CplComp. Superceeded by config if both are specified.

[clock] Set the private clock for this ESMF_CplComp.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.4.11 ESMF_CplCompValidate – Ensure the CplComp is internally consistent

INTERFACE:

```
subroutine ESMF_CplCompValidate(cplcomp, options, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Currently all this method does is to check that the cplcomp exists.
The arguments are:

cplcomp ESMF_CplComp to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.4.12 ESMF_CplCompWriteRestart – Call the CplComp's checkpoint routine

INTERFACE:

```
recursive subroutine ESMF_CplCompWriteRestart(cplcomp, iospec, clock, &
                                              phase, blockingflag, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp), intent(inout) :: cplcomp
type (ESMF_IOSpec), intent(inout), optional :: iospec
type (ESMF_Clock), intent(in), optional :: clock
integer, intent(in), optional :: phase
type (ESMF_BlockingFlag), intent(in), optional :: blockingflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call the associated user checkpoint code for an ESMF_CplComp.
The arguments are:

cplcomp ESMF_CplComp to call writerestart routine for.

[iospec] ESMF_IOSpec object which describes I/O options.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations.

[phase] Component providers must document whether their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For single-phase child components this argument is optional, but if specified it must be ESMF_SINGLEPHASE. For multiple-phase child components, this is the integer phase number to be invoked.

[blockingflag] Valid values are ESMF_BLOCKING (the default) or ESMF_NONBLOCKING.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.4.13 ESMF_CplCompWait - Wait for a CplComp to return

INTERFACE:

```
subroutine ESMF_CplCompWait(cplcomp, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in)            :: cplcomp
integer,            intent(out), optional :: rc
```

DESCRIPTION:

When executing asychronously, wait for an ESMF_CplComp to return.
The arguments are:

cplcomp ESMF_CplComp to wait for.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.5 Class API: SetServices and Related Methods

15.5.1 ESMF_CplCompGetInternalState - Get private data block pointer

INTERFACE:

```
subroutine ESMF_CplCompGetInternalState(cplcomp, dataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(inout) :: cplcomp
type(any), pointer, intent(in) :: dataPointer
integer, intent(out), optional :: rc
```

DESCRIPTION:

Available to be called by an `ESMF_CplComp` at any time after `ESMF_CplCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block, and the address of that block can be registered with the framework and retrieved by this call. When running multiple instantiations of an `ESMF_CplComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMF_CplCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer.
The arguments are:

cplcomp An `ESMF_CplComp` object.

dataPointer A derived type, containing only a pointer to the private data block. The framework will fill in the block and when this call returns the pointer is set to the same address set during `ESMF_CplCompSetInternalState`. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

15.5.2 ESMF_CplCompSetEntryPoint - Set name of CplComp subroutines

INTERFACE:

```
subroutine ESMF_CplCompSetEntryPoint(cplcomp, subroutineType, &
                                     subroutineName, phase, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(inout) :: cplcomp
character(len=*), intent(in) :: subroutineType
subroutine, intent(in) :: subroutineName
integer, intent(in) :: phase
integer, intent(out), optional :: rc
```

DESCRIPTION:

Intended to be called by an `ESMF_CplComp` during the registration process. An `ESMF_CplComp` calls `ESMF_CplCompSetEntryPo` for each of the predefined init, run, and finalize routines, to assocate the internal subroutine to be called for each function. If multiple phases for init, run, or finalize are needed, this can be called with phase numbers.
After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for this child `ESMF_CplComp`.
The arguments are:

cplcomp An ESMF_CplComp object.

subroutineType One of a set of predefined subroutine types - e.g. ESMF_SETINIT, ESMF_SETRUN, ESMF_SETFINAL.

subroutineName The name of the cplcomp subroutine to be associated with the subroutineType. This subroutine does not have to be public to the module.

[phase] For ESMF_CplComps which need to initialize, run, or finalize with mutiple phases, the phase number which corresponds to this subroutine name. For single phase subroutines, either omit this argument, or use the parameter ESMF_SINGLEPHASE. The ESMF_CplComp writer must document the requirements of the ESMF_CplComp for how and when the multiple phases are expected to be called.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.5.3 ESMF_CplCompSetInternalState - Set private data block pointer

INTERFACE:

```
subroutine ESMF_CplCompSetInternalState(cplcomp, dataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(inout) :: cplcomp
type(any), pointer, intent(in) :: dataPointer
integer, intent(out), optional :: rc
```

DESCRIPTION:

Available to be called by an ESMF_CplComp at any time, but expected to be most useful when called during the registration process, or initialization. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block, and the address of that block can be registered with the framework and retrieved by subsequent calls. When running multiple instantiations of an ESMF_CplComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMF_CplCompGetInternalState call retrieves the data pointer. The arguments are:

cplcomp An ESMF_CplComp object.

dataPointer A pointer to the private data block, wrapped in a derived type which contains only a pointer to the block. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

15.5.4 ESMF_CplCompSetServices - Register CplComp interface routines

INTERFACE:

```
subroutine ESMF_CplCompSetServices(cplcomp, subroutineName, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(inout) :: cplcomp
subroutine, intent(in) :: subroutineName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Call an ESMF_CplComp's setservices registration routine. The parent component must first create an ESMF_CplComp, then call this routine. The arguments are the object returned from the create call, plus the public, well-known, subroutine name that is the registration routine for this ESMF_CplComp. This name must be documented by the ESMF_CplComp provider.
After this subroutine returns the framework now knows how to call the initialize, run, and finalize routines for the ESMF_CplComp.
The arguments are:

cplcomp An ESMF_CplComp object.

subroutineName The public name of the cplcomp's ESMF_CplCompSetServices call. An ESMF_CplComp writer must provide this information. Note this is the actual subroutine, not a character string.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16 State Class

16.1 Description

A State contains the data and metadata to be transferred between ESMF components. It is an important class, because it defines a standard for how data is represented in Earth science components. The State construct is a rational compromise between a fully prescribed interface - one that would dictate what specific fields should be transferred between components - and an interface in which data structures are completely ad hoc.
There are two types of States, import and export. An import State contains data that is necessary for a Gridded Component or Coupler Component to execute, and an export State contains the data that a Gridded Component or Coupler Component can make available.
States can contain Bundles, Fields, Arrays, and other States. They cannot contain Fortran arrays.
State methods include creation and deletion, adding and retrieving data items, adding and retrieving attributes, and performing queries.

16.2 State Options

16.2.1 ESMF_StateItemType

DESCRIPTION:
Specifies the type of object being added to or retrieved from an ESMF_State.
Valid values are:

ESMF_STATEITEM_BUNDLE Refers to an ESMF_Bundle within an ESMF_State.

ESMF_STATEITEM_FIELD Refers to an ESMF_Field within an ESMF_State.

ESMF_STATEITEM_ARRAY Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_STATE Refers to an ESMF_State within an ESMF_State.

ESMF_STATEITEM_NAME Refers to a data name used as a placeholder within an ESMF_State.

ESMF_STATEITEM_UNKNOWN Object type within an ESMF_State is unknown.

16.2.2 ESMF_StateType

DESCRIPTION:
Specifies whether an ESMF_State contains data to be imported into a component or exported from a component.
Valid values are:

ESMF_STATE_IMPORT Contains data to be imported into a component.

ESMF_STATE_EXPORT Contains data to be exported out of a component.

ESMF_STATE_INVALID Does not contain valid data.

16.3 Use and Examples

In ESMF applications States are created at the same time as the components that they will be associated with. A Gridded Component generally has one associated import State and one export State. In most cases the States associated with a Gridded Component will be created by the Gridded Component's parent component with no data buffers yet attached. Both the incomplete States and the pointer to the newly created Gridded Component are passed by the parent component into the Gridded Component's initialize method. This is where the States get prepared for use and the import State is first filled with data.
States can be created without the Fields, Arrays, Bundles, and other States they will eventually contain in a number of ways. They can be created with names as placeholders where these data items will eventually be. When the States are passed into the Gridded Component's initialize method, Field, Bundle, and Array create calls can be made in that method to replace the name placeholders with real data objects.
States can also be filled with data items that do not yet have data allocated. Fields, Bundles, and Arrays each have methods that support their creation without actual data allocation - the grid and metadata are set up but no Fortran array of data values is allocated. In this approach, when a State is passed into its associated Gridded component's initialize method, the incomplete Arrays, Fields, and Bundles within the State can allocate or reference data inside the initialize method.
States are passed through the interfaces of the Gridded and Coupler Components' run methods in order to carry data between the components. While we expect a Gridded Component's import State to be filled with data during initialization, its export State will typically be filled over the course of its run method. At the end of a Gridded Component's run method, the filled export State is passed out through the argument list into a Coupler Component's run method. We recommend the convention that it enters the Coupler Component as the Coupler Component's import State. Here is it transformed into a form that another Gridded Component requires, and passed out of the Coupler Component as its export State. It can then be passed into the run method of a recipient Gridded Component as that component's import State.
While the above sounds complicated, the rule is simple: a State going into a component is an import State, and a State leaving a component is an export State.
Data items within a State can be marked needed or not needed, depending on whether they are required for a particular application configuration. If the item is marked not needed, the user can make the Gridded Component's initialize method clever enough to not allocate the data for that item at all and not compute it within the Gridded Component code. For example, some diagnostics may not be desired for all runs.
Other flags will eventually be available for data items within a State, such as data ready for reading or writing, data valid or invalid, and data required for restart or not. These are not yet fully implemented, so only the default value for each value can be set at this time.

```
! !PROGRAM: ESMF_StateEx - State creation and operation
!
! !DESCRIPTION:
!
! This program shows examples of State creation and manipulation
!-----------------------------------------------------------------------------

    ! ESMF Framework module
    use ESMF_Mod
    implicit none

    ! Local variables
    integer :: x, y, rc
    character(ESMF_MAXSTR) :: statename, bundlename, dataname
    type(ESMF_Field) :: field1
```

```
type(ESMF_Bundle) :: bundle1, bundle2
type(ESMF_State) :: state1, state2, state3, state4
```

16.3.1 Empty State Create

Creation of an empty `ESMF_State`, which will be added to later.

```
statename = "Atmosphere"
state1 = ESMF_StateCreate(statename, statetype=ESMF_STATE_IMPORT, rc=rc)
```

16.3.2 Adding Items to a State

Creation of an empty `ESMF_State`, and adding an `ESMF_Bundle` to it. Note that the `ESMF_Bundle` does not get destroyed when the `ESMF_State` is destroyed; the `ESMF_State` only contains a reference to the objects it contains. It also does not make a copy; the original objects can be updated and code accessing them by using the `ESMF_State` will see the updated version.

```
statename = "Ocean"
state2 = ESMF_StateCreate(statename, statetype=ESMF_STATE_EXPORT, rc=rc)


bundlename = "Temperature"
bundle1 = ESMF_BundleCreate(name=bundlename, rc=rc)
print *, "Bundle Create returned", rc


call ESMF_StateAddBundle(state2, bundle1, rc)
print *, "StateAddBundle returned", rc


call ESMF_StateDestroy(state2, rc)


call ESMF_BundleDestroy(bundle1, rc)
```

16.3.3 Adding Placeholders to a State

If a component could potentially produce a large number of optional items, one strategy is to add the names only of those objects to the `ESMF_State`. Other components can call framework routines to set the `ESMF_NEEDED` flag to indicate they require that data. The original component can query this flag and then produce only the data what is required by another component.

```
statename = "Ocean"
state3 = ESMF_StateCreate(statename, statetype=ESMF_STATE_EXPORT, rc=rc)


dataname = "Downward wind"
call ESMF_StateAddNameOnly(state3, dataname, rc)


dataname = "Humidity"
call ESMF_StateAddNameOnly(state3, dataname, rc)
```

16.3.4 Marking an Item Needed

How to set the NEEDED state of an item.

```
dataname = "Downward wind"
call ESMF_StateSetNeeded(state3, dataname, ESMF_NEEDED, rc)
```

16.3.5 Creating a Needed Item

Query an item for the NEEDED status, and creating an item on demand. Similar flags exist for "Ready", "Valid", and "Required for Restart", to mark each data item as ready, having been validated, or needed if the application is to be checkpointed and restarted. The flags are supported to help coordinate the data exchange between components.

```
dataname = "Downward wind"
if (ESMF_StateIsNeeded(state3, dataname, rc)) then


    bundlename = dataname
    bundle2 = ESMF_BundleCreate(name=bundlename, rc=rc)


    call ESMF_StateAddBundle(state3, bundle2, rc)


else
    print *, "Data not marked as needed", trim(dataname)
endif
```

16.4 Restrictions and Future Work

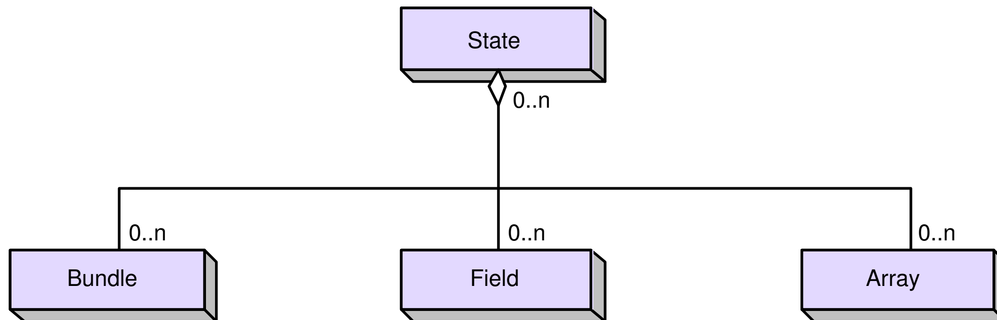
1. **Flags not fully implemented.** The flags for indicating various qualities associated with data items in a State - validity, whether or not the item is required for restart, read/write status - are not fully implemented. Although their defaults can be set, the associated methods for setting and getting these flags have not been implemented.

16.5 Design and Implementation Notes

States contain the name of the associated Component, a flag for Import or Export, and a list of data objects, which can be a combination of Bundles, Fields, and/or Arrays. The objects must be named and have the proper attributes so they can be identified by the receiver of the data. For example, units and other detailed information may need to be associated with the data as an Attribute.

16.6 Object Model

The following is a simplified UML diagram showing the structure of the State class. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



16.7 Class API: Basic State Methods

16.7.1 ESMF_StateAddArray - Add an Array to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddArray()
subroutine ESMF_StateAddOneArray(state, array, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
type(ESMF_Array), intent(in) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a single `array` reference to an existing `state`. The `array` name must be unique within the `state`.
The arguments are:

state An `ESMF_State` object.

array The `ESMF_Array` to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Array` cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.2 ESMF_StateAddArray - Add a list of Arrays to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddArray()
subroutine ESMF_StateAddArrayList(state, arrayCount, arrayList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
integer, intent(in) :: arrayCount
type(ESMF_Array), dimension(:), intent(in) :: arrayList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add multiple `ESMF_Array`s to an `ESMF_State`.
The arguments are:

state An `ESMF_State` object.

arrayCount The number of `ESMF_Array`s to be added.

arrayList The list (Fortran array) of `ESMF_Array`s to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Array`s cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.3 ESMF_StateAddBundle - Add a Bundle to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddBundle()
subroutine ESMF_StateAddOneBundle(state, bundle, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
type(ESMF_Bundle), intent(in) :: bundle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a single `bundle` reference to an existing `state`. The `bundle` name must be unique within the `state`.
The arguments are:

state The `ESMF_State` object.

bundle The `ESMF_Bundle` to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Bundle` cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.4 ESMF_StateAddBundle - Add a list of Bundles to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddBundle()
subroutine ESMF_StateAddBundleList(state, bundleCount, bundleList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
integer, intent(in) :: bundleCount
type(ESMF_Bundle), dimension(:), intent(in) :: bundleList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add multiple `ESMF_Bundle`s to an `ESMF_State`.
The arguments are:

state An `ESMF_State` object.

bundleCount The number of `ESMF_Bundle`s to be added.

bundleList The list (Fortran array) of `ESMF_Bundle`s to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Bundles` cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.5 ESMF_StateAddField - Add a Field to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddField()
subroutine ESMF_StateAddOneField(state, field, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
type(ESMF_Field), intent(in) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a single `field` reference to an existing `state`. The `field` name must be unique within the `state`.
The arguments are:

state An `ESMF_State` object.

field The `ESMF_Field` to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Field` cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.6 ESMF_StateAddField - Add a list of Fields to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddFields()
subroutine ESMF_StateAddFieldList(state, fieldCount, fieldList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
integer, intent(in) :: fieldCount
type(ESMF_Field), dimension(:), intent(in) :: fieldList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add multiple `ESMF_Field`s to an `ESMF_State`.
The arguments are:

state An `ESMF_State` object.

fieldCount The number of `ESMF_Field`s to be added.

fieldList The list (Fortran array) of `ESMF_Field`s to be added. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, the `ESMF_Field`s cannot be safely destroyed before the `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.7 ESMF_StateAddNameOnly - Add a name to a State as a placeholder

INTERFACE:

```
! Private name; call using ESMF_StateAddNameOnly()
subroutine ESMF_StateAddOneName(state, name, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
character (len=*), intent(in) :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add the character string `name` to an existing `state`. It can subsequently be replaced by an actual object with the same name. The `name` must be unique within the `state`. It is available to be marked needed by the consumer of the export `ESMF_State`. Then the data provider can replace the name with the actual `ESMF_Bundle`, `ESMF_Field`, or `ESMF_Array` which carries the needed data.
The arguments are:

state An `ESMF_State` object.

name The name to be added as a placeholder for a data object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.8 ESMF_StateAddNameOnly - Add a list of names to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddNameOnly()
subroutine ESMF_StateAddNameList(state, nameCount, nameList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
integer, intent(in) :: nameCount
character (len=*), intent(in) :: nameList(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a list of names to an existing `state`. They can subsequently be replaced by actual objects with the same name. Each name in the `nameList` must be unique within the `state` It is available to be marked needed by the consumer of the export `ESMF_State`. Then the data provider can replace the name with the actual `ESMF_Bundle`, `ESMF_Field`, or `ESMF_Array` which carries the needed data. Unneeded data need not be generated.
The arguments are:

state An `ESMF_State` object.

nameCount The count of names in the `nameList`.

nameList A list (Fortran array) of character strings to be added as placeholders for data objects.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.9 ESMF_StateAddState - Add a State to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddState()
subroutine ESMF_StateAddOneState(state, nestedState, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
type(ESMF_State), intent(in) :: nestedState
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a `nestedState` reference to an existing `state`. The `nestedState` name must be unique within the container `state`.
The arguments are:

state An `ESMF_State` object. This is the container object.

nestedState The `ESMF_State` to be added. This is the nested object. This is a reference only; when the `ESMF_State` is destroyed the objects contained in it will not be destroyed. Also, nested `ESMF_State`s cannot be safely destroyed before the container `ESMF_State` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.10 ESMF_StateAddState - Add a list of States to a State

INTERFACE:

```
! Private name; call using ESMF_StateAddState()
subroutine ESMF_StateAddStateList(state, nestedStateCount, nestedStateList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
integer, intent(in) :: nestedStateCount
type(ESMF_State), dimension(:), intent(in) :: nestedStateList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add multiple nested `ESMF_States` to a container `ESMF_State`. The nested `ESMF_State` names must be unique within the container `ESMF_State`.
The arguments are:

state An `ESMF_State` object. This is the container object.

nestedStateCount The number of `ESMF_States` to be added.

nestedStateList The list (Fortran array) of `ESMF_States` to be added. This is a reference only; when the container `state` is destroyed the objects contained in it will not be destroyed. Also, the `nestedStateList` cannot be safely destroyed before the container `state` is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.11 ESMF_StateCreate - Create a new State

INTERFACE:

```
function ESMF_StateCreate(stateName, statetype, &
             bundleList, fieldList, arrayList, nestedStateList, &
             nameList, itemCount, &
             neededflag, readyflag, validflag, reqforrestartflag, rc)
```

RETURN VALUE:

```
type(ESMF_State) :: ESMF_StateCreate
```

ARGUMENTS:

```
character(len=*), intent(in), optional :: stateName
type(ESMF_StateType), intent(in), optional :: statetype
type(ESMF_Bundle), dimension(:), intent(in), optional :: bundleList
type(ESMF_Field), dimension(:), intent(in), optional :: fieldList
type(ESMF_Array), dimension(:), intent(in), optional :: arrayList
type(ESMF_State), dimension(:), intent(in), optional :: nestedStateList
character(len=*), dimension(:), intent(in), optional :: nameList
integer, intent(in), optional :: itemCount
type(ESMF_NeededFlag), optional :: neededflag
type(ESMF_ReadyFlag), optional :: readyflag
type(ESMF_ValidFlag), optional :: validflag
type(ESMF_ReqForRestartFlag), optional :: reqforrestartflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a new ESMF_State, set default characteristics for objects added to it, and optionally add initial objects to it. The arguments are:

[stateName] Name of this ESMF_State object. A default name will be generated if none is specified.

[statetype] Import or Export ESMF_State. Valid values are ESMF_STATE_IMPORT, ESMF_STATE_EXPORT, or ESMF_STATE_UNSPECIFIED The default is ESMF_STATE_UNSPECIFIED.

[bundleList] A list (Fortran array) of ESMF_Bundles.

[fieldList] A list (Fortran array) of ESMF_Fields.

[arrayList] A list (Fortran array) of ESMF_Arrays.

[nestedStateList] A list (Fortran array) of ESMF_States to be nested inside the outer ESMF_State.

[nameList] A list (Fortran array) of character string name placeholders.

[itemCount] The total number of things – Bundles, Fields, Arrays, States, and Names – to be added. If itemCount is not specified, it will be computed internally based on the length of each object list. If itemCount is specified this routine will do an error check to verify the total number of items found in the argument lists matches this count of the expected number of items.

[neededflag] Set the default value for new items added to an ESMF_State. Possible values are listed in Section 10.1.6. If not specified, the default value is set to ESMF_NEEDED.

[readyflag] Set the default value for new items added to an ESMF_State. Possible values are listed in Section 10.1.7. If not specified, the default value is set to ESMF_READYTOREAD.

[validflag] Set the default value for new items added to an ESMF_State. Possible values are listed in Section 10.1.10. If not specified, the default value is set to ESMF_VALID.

[reqforrestartflag] Set the default value for new items added to an ESMF_State. Possible values are listed in Section 10.1.9. If not specified, the default value is set to ESMF_REQUIRED_FOR_RESTART.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.12 ESMF_StateDestroy - Release resources for a State

INTERFACE:

```
subroutine ESMF_StateDestroy(state, rc)
```

ARGUMENTS:

```
type(ESMF_State) :: state
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this ESMF_State. ESMF_States contain references only to other objects; when the ESMF_State is destroyed objects contained in it will not be destroyed. Objects inside a ESMF_State cannot be destroyed before the container ESMF_State is destroyed. Since objects can be added to multiple containers, it remains the user's responsibility to manage the destruction of objects when they are no longer in use.
The arguments are:

state Destroy contents of this ESMF_State.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.13 ESMF_StateGet - Get information about a State

INTERFACE:

```
subroutine ESMF_StateGet(state, name, statetype, itemCount, &
                         itemNameList, stateitemtypeList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(out), optional :: name
type(ESMF_StateType), intent(out), optional :: statetype
integer, intent(out), optional :: itemCount
character (len=*), intent(out), optional :: itemNameList(:)
type(ESMF_StateItemType), intent(out), optional :: stateitemtypeList(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the requested information about this ESMF_State.
The arguments are:

state An ESMF_State object to be queried.

[name] Name of this ESMF_State.

[statetype] Import or Export ESMF_State. Possible values are listed in Section 16.2.2.

[itemCount] Count of items in state, including all objects as well as placeholder names.

[itemNameList] Array of item names in state, including placeholder names. itemNameList must be at least itemCount long.

[stateitemtypeList] Array of possible item object types in state, including placeholder names. Must be at least itemCount long. Options are listed in Section 16.2.1.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.14 ESMF_StateGetArray - Retrieve a data Array from a State

INTERFACE:

```
subroutine ESMF_StateGetArray(state, arrayName, array, nestedStateName, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: arrayName
type(ESMF_Array), intent(out) :: array
character (len=*), intent(in), optional :: nestedStateName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an ESMF_Array from an ESMF_State by name. If the ESMF_State contains the object directly, only arrayName is required. If the state contains multiple nested ESMF_States and the object is one level down, this routine can return the object in a single call by specifing the proper nestedStateName. ESMF_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF_States.
The arguments are:

state State to query for an ESMF_Array named arrayName.

arrayName Name of ESMF_Array to be returned.

array Returned reference to the ESMF_Array.

[nestedStateName] Optional. An error if specified when the state argument contains no nested ESMF_States. Required if the state contains multiple nested ESMF_States and the object being requested is in one level down in one of the nested ESMF_State. ESMF_State must be selected by this nestedStateName.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.15 ESMF_StateGetAttribute - Retrieve a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetInt4Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer attribute from the state.
The arguments are:

state An ESMF_State object.

name The name of the attribute to retrieve.

value The 4-byte integer value of the named attribute.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.16 ESMF_StateGetAttribute - Retrieve a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetInt4ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer list attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 4-byte integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.17 ESMF_StateGetAttribute - Retrieve an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetInt8Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

value The 8-byte integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.18 ESMF_StateGetAttribute - Retrieve an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetInt8ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer list attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 8-byte integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.19 ESMF_StateGetAttribute - Retrieve a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetReal4Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

value The 4-byte real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.20 ESMF_StateGetAttribute - Retrieve a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetReal4ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a list of 4-byte real attributes from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 4-byte real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.21 ESMF_StateGetAttribute - Retrieve an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetReal8Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte real attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

value The 8-byte real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.22 ESMF_StateGetAttribute - Retrieve an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetReal8ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a list of 8-byte real attributes from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 8-byte real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.23 ESMF_StateGetAttribute - Retrieve a logical attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetLogicalAttr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

value The logical value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.24 ESMF_StateGetAttribute - Retrieve a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_StateGetAttribute()
subroutine ESMF_StateGetLogicalListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical list attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The logical values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.25 ESMF_StateGetAttribute - Retrieve a character attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_StateGetCharAttr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a character attribute from the `state`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to retrieve.

value The character value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.26 ESMF_StateGetAttributeCount - Query the number of attributes

INTERFACE:

```
subroutine ESMF_StateGetAttributeCount(state, count, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
integer, intent(out) :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the number of attributes associated with the given `state` in the argument `count`.
The arguments are:

state An `ESMF_State` object.

count The number of attributes associated with this object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.27 ESMF_StateGetAttributeInfo - Query State attributes by name

INTERFACE:

```
! Private name; call using ESMF_StateGetAttributeInfo()
subroutine ESMF_StateGetAttrInfoByName(state, name, datatype, &
                                        datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character(len=*), intent(in) :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the named attribute, including `datatype`, `datakind` (if applicable), and item `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to query.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] The number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.28 ESMF_StateGetAttributeInfo - Query State attributes by index number

INTERFACE:

```
! Private name; call using ESMF_StateGetAttributeInfo()
subroutine ESMF_StateGetAttrInfoByNum(state, attributeIndex, name, &
                                      datatype, datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
integer, intent(in) :: attributeIndex
character(len=*), intent(out), optional :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed attribute, including `datatype`, `datakind` (if applicable), and item `count`.
The arguments are:

state An `ESMF_State` object.

attributeIndex The index number of the attribute to query.

name Returns the name of the attribute.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] Returns the number of items in this attribute. For character types, this is the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.29 ESMF_StateGetBundle - Retrieve a Bundle from a State

INTERFACE:

```
subroutine ESMF_StateGetBundle(state, bundleName, bundle, &
                               nestedStateName, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: bundleName
type(ESMF_Bundle), intent(out) :: bundle
character (len=*), intent(in), optional :: nestedStateName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an ESMF_Bundle from an ESMF_State by name. If the ESMF_State contains the object directly, only bundleName is required. If the state contains multiple nested ESMF_States and the object is one level down, this routine can return the object in a single call by specifing the proper nestedStateName. ESMF_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF_States.
The arguments are:

state State to query for a ESMF_Bundle named bundlename.

bundleName Name of ESMF_Bundle to be returned.

bundle Returned reference to the ESMF_Bundle.

[nestedStateName] Optional. An error if specified when the state argument contains no nested ESMF_States. Required if the state contains multiple nested ESMF_States and the object being requested is in one level down in one of the nested ESMF_State. ESMF_State must be selected by this nestedStateName.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.30 ESMF_StateGetField - Retrieve a Field from a State

INTERFACE:

```
subroutine ESMF_StateGetField(state, fieldName, field, &
                              nestedStateName, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: fieldName
type(ESMF_Field), intent(out) :: field
character (len=*), intent(in), optional :: nestedStateName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an ESMF_Field from an ESMF_State by name. If the ESMF_State contains the object directly, only fieldname is required. If the state contains multiple nested ESMF_States and the object is one level down, this routine can return the object in a single call by specifing the proper nestedStateName. ESMF_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF_States.
The arguments are:

state State to query for an ESMF_Field named fieldName.

fieldName Name of ESMF_Field to be returned.

field Returned reference to the ESMF_Field.

[nestedStateName] Optional. An error if specified when the state argument contains no nested ESMF_States. Required if the state contains multiple nested ESMF_States and the object being requested is in one level down in one of the nested ESMF_State. ESMF_State must be selected by this nestedStateName.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.31 ESMF_StateGetNeeded - Query whether a data item is needed

INTERFACE:

```
subroutine ESMF_StateGetNeeded(state, itemName, neededflag, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
type(ESMF_NeededFlag), intent(out) :: neededflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the status of the neededflag for the data item named by itemName in the ESMF_State.
The arguments are:

state The ESMF_State to query.

itemName Name of the data item to query.

neededflag Whether state item is needed or not for a particular application configuration. Possible values are listed in Section 10.1.6.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.32 ESMF_StateGetState - Retrieve a State nested in a State

INTERFACE:

```
subroutine ESMF_StateGetState(state, nestedStateName, nestedState, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: nestedStateName
type(ESMF_State), intent(out) :: nestedState
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a nested ESMF_State from another ESMF_State by name. This does not allow the caller to retrieve an ESMF_State from two levels down. It returns immediate child objects only.
The arguments are:

state The ESMF_State to query for a nested ESMF_State named stateName.

nestedStateName Name of nested ESMF_State to return.

nestedState Returned ESMF_State.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.33 ESMF_StateIsNeeded – Return logical true if data item needed

INTERFACE:

```
function ESMF_StateIsNeeded(state, itemName, rc)
```

RETURN VALUE:

```
logical :: ESMF_StateIsNeeded
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns true if the status of the needed flag for the data item named by itemName in the ESMF_State is ESMF_STATEITEM_NEEDED. Returns false for no item found with the specified name or item marked not needed. Also sets error code if dataname not found.
The arguments are:

state ESMF_State to query.

itemName Name of the data item to query.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.34 ESMF_StatePrint - Print the internal data for a State

INTERFACE:

```
subroutine ESMF_StatePrint(state, options, rc)
```

ARGUMENTS:

```
type(ESMF_State) :: state
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the state to stdout.
The arguments are:

state The ESMF_State to print.

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

16.7.35 ESMF_StateSetAttribute - Set a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetInt4Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The 4-byte integer value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.36 ESMF_StateSetAttribute - Set a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetInt4ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer list attribute to the `state`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

count The number of integers in the `valueList`.

valueList The 4-byte integer values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.37 ESMF_StateSetAttribute - Set an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetInt8Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte integer attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The 8-byte integer value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.38 ESMF_StateSetAttribute - Set an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetInt8ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte integer list attribute to the `state`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

count The number of integers in the `valueList`.

valueList The 8-byte integer values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.39 ESMF_StateSetAttribute - Set a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetReal4Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The 4-byte real value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.40 ESMF_StateSetAttribute - Set a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetReal4ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real list attribute to the `state`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

count The number of reals in the `valueList`.

value The 4-byte real values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.41 ESMF_StateSetAttribute - Set an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetReal8Attr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The 8-byte real value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.42 ESMF_StateSetAttribute - Set an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetReal8ListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real list attribute to the `state`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

count The number of reals in the `valueList`.

value The 8-byte real values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.43 ESMF_StateSetAttribute - Set a logical attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetLogicalAttr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The logical true/false value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.44 ESMF_StateSetAttribute - Set a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetLogicalListAttr(state, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical list attribute to the `state`. The attribute has a `name` and a `valueList`. The number of logical items in the `valueList` is given by `count`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

count The number of logicals in the `valueList`.

valueList The logical true/false values of the attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.45 ESMF_StateSetAttribute - Set a character attribute

INTERFACE:

```
! Private name; call using ESMF_StateSetAttribute()
subroutine ESMF_StateSetCharAttr(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a character attribute to the `state`. The attribute has a `name` and a `value`.
The arguments are:

state An `ESMF_State` object.

name The name of the attribute to set.

value The character value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.46 ESMF_StateSetNeeded - Set if a data item is needed

INTERFACE:

```
subroutine ESMF_StateSetNeeded(state, itemName, neededflag, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
character (len=*), intent(in) :: itemName
type(ESMF_NeededFlag), intent(in) :: neededflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the status of the `needed` flag for the data item named by `itemName` in the `ESMF_State`.
The arguments are:

state The `ESMF_State` to set.

itemName Name of the data item to set.

neededflag Set status of data item to this. See Section 10.1.6 for possible values.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.7.47 ESMF_StateValidate - Check validity of a State

INTERFACE:

```
subroutine ESMF_StateValidate(state, options, rc)
```

ARGUMENTS:

```
type(ESMF_State) :: state
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `state` is internally consistent. Currently this method determines if the `state` is uninitialized or already destroyed. The method returns an error code if problems are found.
The arguments are:

state The `ESMF_State` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.8 Class API: State Overloads for Fortran Arrays

16.8.1 ESMF_StateGetDataPointer - Retrieve Fortran pointer directly from a State

INTERFACE:

```
! Private name; call using ESMF_StateGetDataPointer()
subroutine ESMF_StateGetDataPointer<rank><type><kind>(state, itemName,
dataPointer, copyflag, nestedStateName, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character(len=*), intent(in) :: itemName
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: dataPointer
type(ESMF_CopyFlag), intent(in), optional :: copyflag
character(len=*), intent(in), optional :: nestedStateName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Retrieves data from a state, returning a direct Fortran pointer to the data array. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

state The `ESMF_State` to query.

itemName The name of the Bundle, Field, or Array to return data from.

dataPointer An unassociated Fortran pointer of the proper Type, Kind, and Rank as the data in the State. When this call returns successfully, the pointer will now reference the data in the State. This is either a reference or a copy, depending on the setting of the following argument. The default is to return a reference.

[copyflag] Defaults to ESMF_DATA_REF. If set to ESMF_DATA_COPY, a separate copy of the data will be made and the pointer will point at the copy.

[nestedStateName] Optional. If multiple states are present, a specific state name must be given.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

Part III
Infrastructure: Fields and Grids

17 Overview of Infrastructure Data Handling

The ESMF infrastructure data classes are part of the framework's hierarchy of structures for handling Earth system model data and metadata on parallel platforms. The hierarchy is in complexity; the simplest data class in the infrastructure represents a distributed array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. Data class methods are called both from user-written code and from other classes internal to the framework.
Data classes are distributed over **DE**s, or **Decomposition Elements**. A DE represents a piece of a decomposition. A DELayout is a collection of DEs with some associated connectivity that describes a specific distribution. For example, the distribution of a grid divided into four segments in the x-dimension would be expressed in ESMF as a DELayout with four DEs lying along an x-axis. This abstract concept enables a data decomposition to be defined in terms of threads, MPI processes, virtual decomposition elements, or combinations of these without changes to user code. This is a primary strategy for ensuring optimal performance and portability for codes using the ESMF for communications.
ESMF data classes are useful because they provide a standard, convenient way for developers to collect together information related to model or observational data. The information assembled in a data class includes a data pointer, a set of attributes (e.g. units, although attributes can also be user-defined), and a description of an associated grid. The same set of information within an ESMF data object can be used by the framework to arrange intercomponent data transfers, to perform I/O, for communications such as gathers and scatters, for simplification of interfaces within user code, for debugging, and for other functions. This unifies and organizes codes overall so that the user need not define different representations of metadata for the same field for I/O and for component coupling.
Since it is critical that users be able to introduce ESMF into their codes easily and incrementally, ESMF data classes can be created based on native Fortran pointers. Likewise, there are methods for retrieving native Fortran pointers from within ESMF data objects. This allows the user to perform allocations using ESMF, and to retrieve Fortran arrays later for optimized model calculations. The ESMF data classes do not have associated differential operators or other mathematical methods.
For flexibility, it is not necessary to build an ESMF data object all at once. For example, it's possible to create a field but to defer allocation of the associated field data until a later time.

Key Features
Hierarchy of data structures designed specifically for the Earth system domain and high performance, parallel computing.
Multi-use ESMF structures simplify user code overall.
Data objects support incremental construction and deferred allocation.
Native Fortran arrays can be associated with or retrieved from ESMF data objects, for ease of adoption, convenience, and performance.

17.1 Infrastructure Data Classes

The main classes that are used for model and observational data manipulation are as follows:

- **Array** An ESMF Array contains a data pointer, information about its associated datatype, precision, and dimension.

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation. ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

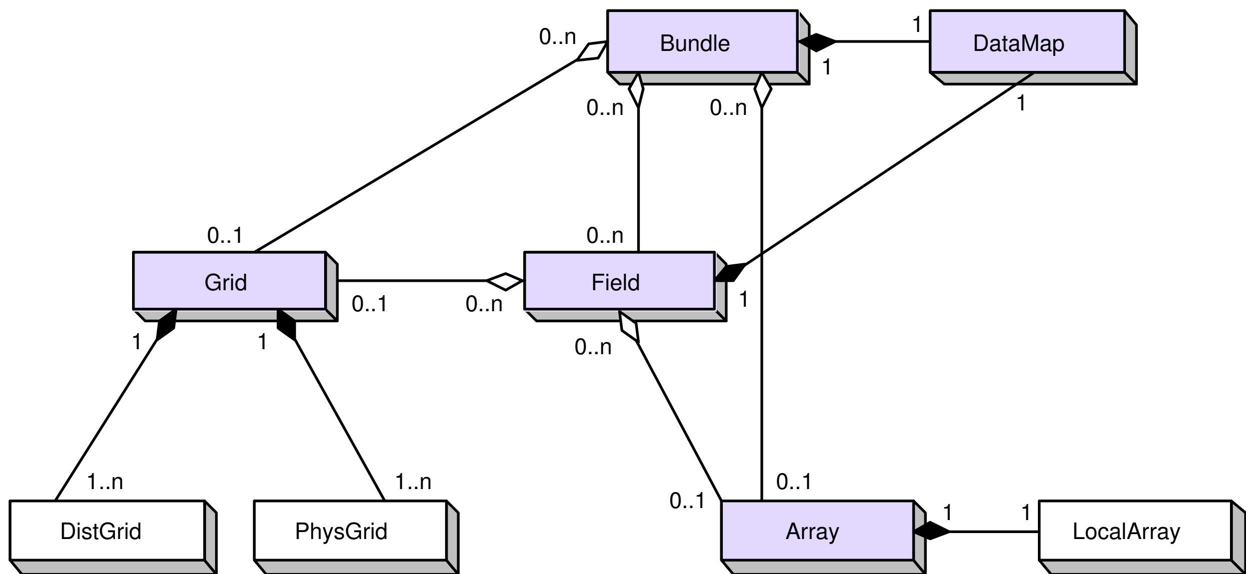
- **Field** A Field holds model and/or observational data together with its underlying grid or set of spatial locations. It provides methods for configuration, initialization, setting and retrieving data values, data I/O, data regridding, and manipulation of attributes.

- **Bundle** Groups of Fields on the same underlying physical grid can be collected into a single object called a Bundle. A Bundle provides two major functions: it allows groups of Fields to be manipulated using a single identifier, for example during export or import of data between Components; and it allows data from multiple Fields to be packed together in memory for higher locality of reference and ease in subsetting operations. Packing a set of Fields into a single Bundle before performing a data communication allows the set to be transferred at once rather than as a Field at a time. This can improve performance on high-latency platforms.

  Bundle objects contain methods for setting and retrieving constituent fields, regridding, data I/O, and reordering of data in memory.

17.2 Object Model

The following is a simplified UML diagram showing the relationships among ESMF Field, Grid and Bundle classes. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



17.3 Design and Implementation Notes

1. In communication methods such as Regrid, Redist, Scatter, etc. the Bundle and Field code cascades down through the Array code, so that the actual computations exist in only one place in the source.

18 Bundle Class

18.1 Description

The Bundle class represents "bundles" of Fields that are discretized on the same Grid and distributed in the same manner. Fields within a Bundle may be located at different locations relative to the vertices of their common Grid. The Fields in a Bundle may be of different dimensions, as long as the Grid dimensions that are distributed are the same. For example, a surface Field on a distributed lat/lon Grid and a 3D Field with an added vertical dimension on the same distributed lat/lon Grid can be included in the same Bundle.
Bundles currently function mainly as convenient containers for storing Fields. Bundles can be created and destroyed, can have attributes added or retrieved, and can have Fields added or retrieved. Methods include a variety of queries that return information about the attributes and the Fields that a Bundle contains. The Fortran data pointer of a Field within a Bundle can be obtained by passing the Bundle a Field name.
Memory layout information is stored in a BundleDataMap object which is attached to the Bundle. It can be accessed by querying the Bundle. Although we have made the BundleDataMap public, many of the memory layout options have not been implemented.
Bundles are one of the data objects that can be added to States, which are used for sending to or receiving data from other components.
In the future Bundles will serve as a mechanism for performance optimization. ESMF will take advantage of the similarities of the Fields within a Bundle in order to implement collective communication, IO, and regridding. See Section 18.4 for a description of features that are being planned.

18.2 Bundle Options

18.2.1 ESMF_PackFlag

DESCRIPTION:
Specifies whether a Bundle is packed or not. A packed Bundle contains an array in which all the data in its constituent Fields is packed contiguously. Bundles that are not packed are not guaranteed to carry a contiguous array of their data. This flag is not yet implemented; the value is always set to `ESMF_NO_PACKED_DATA`.
Valid values are:

ESMF_PACKED_DATA Contains a packed array.

ESMF_NO_PACKED_DATA Does not contain a packed array.

18.3 Use and Examples

Examples of creating, destroying and accessing Bundles and their constituent Fields are provided in this section, along with some notes on Bundle methods.

18.3.1 Bundle Creation

After creating multiple Fields, a Bundle can be created by passing a list of the Fields into the method `ESMF_BundleCreate()`. The Bundle will contain references to the Fields. An empty Bundle can also be created and Fields added singularly or in groups.
The feature which requests a packed Array be created from the combined Field data arrays is not implemented in this version of the framework.

18.3.2 Accessing Bundle Data

To access data in a Bundle the user can provide a Field name and retrieve the Field's Fortran data pointer. Alternatively, the user can retrieve the data in the form of an ESMF Field and use the Field-level interfaces.
The packed Array feature of Bundles is not implemented in this version of the Framework.

18.3.3 Bundle Deletion

The user must call `ESMF_BundleDestroy()` before deleting any of the Fields it contains. Because Fields can be shared by multiple Bundles and States, they are not deleted by this call.
See the following code fragments for examples of how to create new Bundles.

```
! Example program showing various ways to create a Bundle object.

program ESMF_BundleCreateEx

! ESMF Framework module
use ESMF_Mod

implicit none

! Local variables
integer :: i, x, y, rc, mycell, fieldcount
type(ESMF_Grid) :: grid
type(ESMF_ArraySpec) :: arrayspec
type(ESMF_FieldDataMap) :: datamap
type(ESMF_DELayout) :: delayout
type(ESMF_VM) :: vm
character (len = ESMF_MAXSTR) :: bname1, bname2, fname1, fname2
type(ESMF_IOSpec) :: iospec
type(ESMF_Field) :: field(10), returnedfield1, returnedfield2, simplefield
type(ESMF_Bundle) :: bundle1, bundle2, bundle3
real (selected_real_kind(6,45)), dimension(:,:), pointer :: f90ptr1, f90ptr2
integer :: counts(2)
real(ESMF_KIND_R8) :: min_coord(2), max_coord(2)
```

```
!   !  Create several Fields and add them to a new Bundle.

    counts = (/ 100, 200 /)
    min_coord = (/  0.0,  0.0 /)
    max_coord = (/ 50.0, 60.0 /)
    delayout = ESMF_DELayoutCreate(vm, rc=rc)
    grid = ESMF_GridCreateHorzXYUni(counts, min_coord, max_coord, &
                horzStagger=ESMF_GRID_HORZ_STAGGER_A, rc=rc)
    call ESMF_GridDistribute(grid, delayout=delayout, rc=rc)



    call ESMF_ArraySpecSet(arrayspec, 2, ESMF_DATA_REAL, ESMF_R8, rc)
    field(1) = ESMF_FieldCreate(grid, arrayspec, &
                                horzRelloc=ESMF_CELL_CENTER, &
                                name="pressure", rc=rc)


    field(2) = ESMF_FieldCreate(grid, arrayspec, &
                                horzRelloc=ESMF_CELL_CENTER, &
                                name="temperature", rc=rc)


    field(3) = ESMF_FieldCreate(grid, arrayspec, &
```

```
                                horzRelloc=ESMF_CELL_CENTER, &
                                name="heat flux", rc=rc)

    bundle1 = ESMF_BundleCreate(3, field, name="atmosphere data", rc=rc)

    print *, "Bundle example 1 returned"

!-------------------------------------------------------------------------
!   !  Create an empty Bundle and then add a single field to it.


    simplefield = ESMF_FieldCreate(grid, arrayspec, &
                                horzRelloc=ESMF_CELL_CENTER, name="rh", rc=rc)


    bundle2 = ESMF_BundleCreate(name="time step 1", rc=rc)


    call ESMF_BundleAddField(bundle2, simplefield, rc)


    call ESMF_BundleGet(bundle2, fieldCount=fieldcount, rc=rc)

    print *, "Bundle example 2 returned, fieldcount =", fieldcount

!-------------------------------------------------------------------------
!   !  Create an empty Bundle and then add multiple fields to it.


    bundle3 = ESMF_BundleCreate(name="southern hemisphere", rc=rc)


    call ESMF_BundleAddField(bundle3, 3, field, rc)


    call ESMF_BundleGet(bundle3, fieldCount=fieldcount, rc=rc)

    print *, "Bundle example 3 returned, fieldcount =", fieldcount

!-------------------------------------------------------------------------
!   !  Get a Field back from a Bundle, first by name and then by index.
!   !  Also get the Bundle name.

    call ESMF_BundleGetField(bundle1, "pressure", returnedfield1, rc)


    call ESMF_FieldGet(returnedfield1, name=fname1, rc=rc)


    call ESMF_BundleGetField(bundle1, 2, returnedfield2, rc)


    call ESMF_FieldGet(returnedfield2, name=fname2, rc=rc)
```

```
    call ESMF_BundleGet(bundle1, name=bname1, rc=rc)

    print *, "Bundle example 4 returned, field names = ", &
                   trim(fname1), ", ", trim(fname2)
    print *, "Bundle name = ", trim(bname1)


!-------------------------------------------------------------------------

     call ESMF_BundleDestroy(bundle1, rc=rc)


     call ESMF_BundleDestroy(bundle2, rc=rc)


     call ESMF_BundleDestroy(bundle3, rc=rc)


     do i=1, 3
         call ESMF_FieldDestroy(field(i),rc=rc)


     enddo

     call ESMF_FieldDestroy(simplefield, rc=rc)


     end program ESMF_BundleCreateEx
```

18.4 Restrictions and Future Work

1. **No mathematical operators.** The Bundle class does not support differential or other mathematical operators. We do not anticipate providing this functionality in the near future.

2. **Limited validation and options.** We are planning to increase the number of validity checks available for Bundles as soon as possible, foremost the ability to check that the Fields it contains are on the same Grid. We also will be working on print options.

3. **Limited collective operations.** The Bundle class does not automatically support collective field operations. In order to perform the same operation on Fields within a Bundle, you must currently loop over a query for each Field and call the method on each Field in turn. We expect to have interfaces for non-optimized collective operations such as regridding, data communication, and IO shortly.

   One of the options that we are currently working on for Bundles is packing. Packing means that the data from all the Fields that comprise the Bundle are copied into a single Array and manipulated collectively. This operation can be done without destroying the original Field data. Packing is being designed to facilitate optimized regridding, data communication, and IO operations. It will be possible to collectively manipulate all the Fields within a Bundle at once, rather than operating on each Field separately. This will reduce the latency overhead of the communication.

4. **Interleaving Fields within a Bundle.** Data locality is important for performance on some computing platforms. An interleave option will allow the user to create a packed Bundle in which Fields are either concatenated in memory or in which Field elements are interleaved.

18.5 Design and Implementation Notes

1. **Fields in a Bundle reference the same Grid.** In order to reduce memory requirements and ensure consistency, the Fields within a Bundle all reference the same Grid object.

18.6 Class API: Basic Bundle Methods

18.6.1 ESMF_BundleAddField - Add a Field to a Bundle

INTERFACE:

```
! Private name; call using ESMF_BundleAddField()
subroutine ESMF_BundleAddOneField(bundle, field, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
type(ESMF_Field), intent(in) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Adds a single `field` to an existing `bundle`. The `field` must be associated with the same `ESMF_Grid` as the other `ESMF_Fields` in the `bundle`. The `field` is referenced by the `bundle`, not copied.
The arguments are:

bundle The `ESMF_Bundle` to add the `ESMF_Field` to.

field The `ESMF_Field` to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.2 ESMF_BundleAddField - Add a list of Fields to a Bundle

INTERFACE:

```
! Private name; call using ESMF_BundleAddField()
subroutine ESMF_BundleAddFieldList(bundle, fieldCount, fieldList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
integer, intent(in) :: fieldCount
type(ESMF_Field), dimension(:), intent(in) :: fieldList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Adds a `fieldList` to an existing `ESMF_Bundle`. The items added from the `ESMF_fieldList` must be associated with the same `ESMF_Grid` as the other `ESMF_Fields` in the `bundle`. The items in the `fieldList` are referenced by the `bundle`, not copied.
The arguments are:

bundle `ESMF_Bundle` to add `ESMF_Fields` to.

fieldCount Number of `ESMF_Fields` to be added to the `ESMF_Bundle`; must be equal to or less than the number of items in the `fieldList`.

fieldList Array of existing `ESMF_Fields`. The first `fieldCount` items will be added to the `ESMF_Bundle`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.3 ESMF_BundleCreate - Create a Bundle from existing Fields

INTERFACE:

```
! Private name; call using ESMF_BundleCreate()
function ESMF_BundleCreateNew(fieldCount, fieldList, &
                              packflag, bundleinterleave, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Bundle) :: ESMF_BundleCreateNew
```

ARGUMENTS:

```
integer, intent(in) :: fieldCount
type(ESMF_Field), dimension (:) :: fieldList
type(ESMF_PackFlag), intent(in), optional :: packflag
type(ESMF_InterleaveFlag), intent(in), optional :: bundleinterleave
character (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

Creates an ESMF_Bundle from a list of existing ESMF_Fields stored in a fieldList. All items in the fieldList must be associated with the same ESMF_Grid. Returns a new ESMF_Bundle.
The arguments are:

fieldCount Number of fields to be added to the new ESMF_Bundle. Must be equal to or less than the number of ESMF_Fields in the fieldList.

fieldList Array of existing ESMF_Fields. The first ESMF_FieldCount items will be added to the new ESMF_Bundle.

[packflag] The packing option is not yet implemented. See Section 18.4 for a description of packing, and Section 18.2.1 for anticipated values. The current implementation corresponds to the value ESMF_NO_PACKED_DATA, which means that every ESMF_Field is referenced separately rather than being copied into a single contiguous buffer. This is the case no matter what value, if any, is passed in for this argument.

[bundleinterleave] The interleave option is not yet implemented. See Section 18.4 for a brief description of interleaving, and Section **??** for anticipated values. The flag is not applicable to the current implementation, since it applies only to packed data (see the packflag argument).

[name] ESMF_Bundle name. A default name is generated if one is not specified.

[iospec] The ESMF_IOSpec is not yet used by ESMF_Bundles. Any values passed in will be ignored.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

18.6.4 ESMF_BundleCreate - Create a Bundle with no Fields

INTERFACE:

```
! Private name; call using ESMF_BundleCreate()
function ESMF_BundleCreateNoFields(grid, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Bundle) :: ESMF_BundleCreateNoFields
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in), optional :: grid
character (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

Creates an `ESMF_Bundle` with no associated `ESMF_Fields`.
The arguments are:

[grid] The `ESMF_Grid` which all `ESMF_Fields` added to this `ESMF_Bundle` must be associated with. If not specified now, the grid associated with the first `ESMF_Field` added will be used as the reference grid for the `ESMF_Bundle`.

[name] `ESMF_Bundle` name. A default name is generated if one is not specified.

[iospec] The `ESMF_IOSpec` is not yet used by `ESMF_Bundles`. Any values passed in will be ignored.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.5 ESMF_BundleDestroy - Free all resources associated with a Bundle

INTERFACE:

```
subroutine ESMF_BundleDestroy(bundle, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle) :: bundle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases resources associated with the `bundle`. This method does not destroy the `ESMF_Fields` that the `bundle` contains. The `bundle` should be destroyed before the `ESMF_Fields` within it are.

bundle An `ESMF_Bundle` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.6 ESMF_BundleGet - Return information about a Bundle

INTERFACE:

```
subroutine ESMF_BundleGet(bundle, grid, fieldCount, name, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
type(ESMF_Grid), intent(out), optional :: grid
integer, intent(out), optional :: fieldCount
character (len = *), intent(out), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information about the `bundle`. If the `ESMF_Bundle` was originally created without specifying a name, a unique name will have been generated by the framework.
The arguments are:

bundle The `ESMF_Bundle` object to query.

[grid] The `ESMF_Grid` associated with the `bundle`.

[fieldCount] Number of `ESMF_Field`s in the `bundle`.

[name] A character string where the `bundle` name is returned.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.7 ESMF_BundleGetAttribute - Retrieve a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetInt4Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.8 ESMF_BundleGetAttribute - Retrieve a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetInt4ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an integer list attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

count The number of values in the list.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.9 ESMF_BundleGetAttribute - Retrieve an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetInt8Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.10 ESMF_BundleGetAttribute - Retrieve an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetInt8ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer list attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

count The number of values in the list.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.11 ESMF_BundleGetAttribute - Retrieve a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetReal4Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.12 ESMF_BundleGetAttribute - Retrieve a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetReal4ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real list attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

count The number of values in the list.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.13 ESMF_BundleGetAttribute - Retrieve an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetReal8Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte real attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.14 ESMF_BundleGetAttribute - Retrieve an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetReal8ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte real list attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

count The number of values in the list.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.15 ESMF_BundleGetAttribute - Retrieve a logical attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetLogicalAttr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The logical value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.16 ESMF_BundleGetAttribute - Retrieve a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetLogicalListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical list attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

count The number of values in the list.

valueList The logical values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.17 ESMF_BundleGetAttribute - Retrieve a character attribute

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttribute()
subroutine ESMF_BundleGetCharAttr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a character attribute from the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to retrieve.

value The character value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.18 ESMF_BundleGetAttributeCount - Query the number of attributes

INTERFACE:

```
subroutine ESMF_BundleGetAttributeCount(bundle, count, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
integer, intent(out) :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the number of attributes associated with the given `bundle` in the argument `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

count The number of attributes associated with this object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.19 ESMF_BundleGetAttributeInfo - Query Bundle attributes by name

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttributeInfo()
subroutine ESMF_BundleGetAttrInfoByName(bundle, name, datatype, &
                                        datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character(len=*), intent(in) :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the named attribute, including `datatype`, `datakind` (if applicable), and item `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to query.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] The number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

18.6.20 ESMF_BundleGetAttributeInfo - Query Bundle attributes by index number

INTERFACE:

```
! Private name; call using ESMF_BundleGetAttributeInfo()
subroutine ESMF_BundleGetAttrInfoByNum(bundle, attributeIndex, name, &
                                       datatype, datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
integer, intent(in) :: attributeIndex
character(len=*), intent(out), optional :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed attribute, including datatype, datakind (if applicable), and item count.
The arguments are:

bundle An ESMF_Bundle object.

attributeIndex The index number of the attribute to query.

name Returns the name of the attribute.

[datatype] The data type of the attribute. One of the values ESMF_DATA_INTEGER, ESMF_DATA_REAL, ESMF_DATA_LOGICAL, or ESMF_DATA_CHARACTER.

[datakind] The datakind of the attribute, if attribute is type ESMF_DATA_INTEGER or ESMF_DATA_REAL. One of the values ESMF_I4, ESMF_I8, ESMF_R4, or ESMF_R8. For all other types the value ESMF_NOKIND is returned.

[count] Returns the number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

18.6.21 ESMF_BundleGetField - Retrieve a Field by name

INTERFACE:

```
! Private name; call using ESMF_BundleGetField()
subroutine ESMF_BundleGetFieldByName(bundle, name, field, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
type(ESMF_Field), intent(out) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a `field` from a `bundle` using the `field`'s `name`.
The arguments are:

bundle `ESMF_Bundle` to query for `ESMF_Field`.

name `ESMF_Field` name.

field Returned `ESMF_Field`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.22 ESMF_BundleGetField - Retrieve a Field by index number

INTERFACE:

```
! Private name; call using ESMF_BundleGetField()
subroutine ESMF_BundleGetFieldByNum(bundle, fieldIndex, field, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
integer, intent(in) :: fieldIndex
type(ESMF_Field), intent(out) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a `field` from a `bundle` by index number.
The arguments are:

bundle `ESMF_Bundle` to query for `ESMF_Field`.

fieldIndex `ESMF_Field` index number; first `fieldIndex` is 1.

field Returned `ESMF_Field`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.23 ESMF_BundlePrint - Print information about a Bundle

INTERFACE:

```
subroutine ESMF_BundlePrint(bundle, options, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len=*), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints diagnostic information about the `bundle` to `stdout`.
The arguments are:

bundle An `ESMF_Bundle` object.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.24 ESMF_BundleSetAttribute - Set a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetInt4ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer list attribute to the `bundle`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.25 ESMF_BundleSetAttribute - Set an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetInt8Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte integer attribute to the `bundle`. The attribute has a `name` and a `value`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

value The integer value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.26 ESMF_BundleSetAttribute - Set an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetInt8ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer list attribute to the `bundle`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.27 ESMF_BundleSetAttribute - Set a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetReal4Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real attribute to the `bundle`. The attribute has a `name` and a `value`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

value The real value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.28 ESMF_BundleSetAttribute - Set an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetReal8Attr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real attribute to the `bundle`. The attribute has a `name` and a `value`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

value The real value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.29 ESMF_BundleSetAttribute - Set a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetReal4ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real list attribute to the `bundle`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

count The number of reals in the `valueList`.

value The real values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.30 ESMF_BundleSetAttribute - Set an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetReal8ListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real list attribute to the `bundle`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

count The number of reals in the `valueList`.

value The real values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.31 ESMF_BundleSetAttribute - Set a logical attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetLogicalAttr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical attribute to the `bundle`. The attribute has a `name` and a `value`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

value The logical true/false value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.32 ESMF_BundleSetAttribute - Set a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetLogicalListAttr(bundle, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical list attribute to the `bundle`. The attribute has a `name` and a `valueList`. The number of logical items in the `value` list is given by `count`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

count The number of logicals in the `valueList`.

valueList The logical values of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.33 ESMF_BundleSetAttribute - Set a character attribute

INTERFACE:

```
! Private name; call using ESMF_BundleSetAttribute()
subroutine ESMF_BundleSetCharAttr(bundle, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a character attribute to the `bundle`. The attribute has a `name` and a `value`.
The arguments are:

bundle An `ESMF_Bundle` object.

name The name of the attribute to set.

value The character value of the attribute to set.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.34 ESMF_BundleSetGrid - Associate a Grid with an empty Bundle

INTERFACE:

```
subroutine ESMF_BundleSetGrid(bundle, grid, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
type(ESMF_Grid), intent(in) :: grid
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the `grid` for a `bundle` that contains no `ESMF_Field`s. All `ESMF_Field`s added to this `bundle` must be associated with the same `ESMF_Grid`. Returns an error if there is already an `ESMF_Grid` associated with the `bundle`.
The arguments are:

bundle An `ESMF_Bundle` object.

grid The `ESMF_Grid` which all `ESMF_Field`s added to this `ESMF_Bundle` must have.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

18.6.35 ESMF_BundleValidate - Check validity of a Bundle

INTERFACE:

```
subroutine ESMF_BundleValidate(bundle, options, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character (len=*), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `bundle` is internally consistent. Currently this method determines if the `bundle` is uninitialized or already destroyed. The method returns an error code if problems are found.
The arguments are:

bundle `ESMF_Bundle` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if the `bundle` is valid.

18.7 Class API: Bundle Overloads for Fortran Arrays

18.7.1 ESMF_BundleGetDataPointer - Retrieve Fortran pointer directly from a Bundle

INTERFACE:

```
! Private name; call using ESMF_BundleGetDataPointer()
subroutine ESMF_BundleGetDataPointer<rank><type><kind>(bundle, &
fieldName, dataPointer, copyflag, rc)
```

ARGUMENTS:

```
type(ESMF_Bundle), intent(in) :: bundle
character(len=*), intent(in) :: fieldName
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: dataPointer
type(ESMF_CopyFlag), intent(in), optional :: copyflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Retrieves data from the `bundle`, returning a direct Fortran pointer to the data. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

bundle The `ESMF_Bundle` to query.

fieldName The name of the `ESMF_Field` inside the `bundle` to return. The `bundle` cannot have packed data.

dataPointer An unassociated Fortran pointer of the proper Type, Kind, and Rank as the data in the Bundle. When this call returns successfully, the pointer will now point to the data in the Bundle. This is either a reference or a copy, depending on the setting of the following argument. The default is to return a reference.

[copyflag] Defaults to `ESMF_DATA_REF`. If set to `ESMF_DATA_COPY`, a separate copy of the data will be made and the pointer will point at the copy.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

19 BundleDataMap Class

19.1 Description

The BundleDataMap class specifies how the Fields within a packed Bundle are interleaved. In a packed Bundle, the data arrays of constituent Fields have been copied or transferred to a single combined data array. This is generally done for optimization - either to increase data locality for quick data retrieval from memory or to aggregate communications to reduce latency.
Packed Bundles are not fully implemented. Currently the Field data within Bundles are always stored as individual data references. At this point the BundleDataMap class is a placeholder; values for a Bundle interleave flag can be set and retrieved, but they are not used by the framework.

19.2 BundleDataMap Options

19.3 Use and Examples

A BundleDataMap is a shallow object. It can simply be declared as a local (stack) variable, and does not have to be created or destroyed. To initialize a BundleDataMap with default values a set default method is provided. To alter or query an existing object, use the set and get methods. A print method is provided for human-readable output or debugging.

```
! !PROGRAM: ESMF_BundleDataMapEx - BundleDataMap manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of BundleDataMap set and get usage
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! local variables
      type(ESMF_BundleDataMap) :: bundleDM
      type(ESMF_InterleaveFlag) :: il


      ! return code
      integer:: rc

      ! initialize ESMF framework
      call ESMF_Initialize(rc=rc)
```

19.3.1 Setting BundleDataMap Defaults

This example shows how to set the default values in an `ESMF_BundleDataMap`.

```
      call ESMF_BundleDataMapSetDefault(bundleDM, rc=rc)

      print *, "Default values for BundleDataMap = "
      call ESMF_BundleDataMapPrint(bundleDM, rc=rc)
```

19.3.2 Setting BundleDataMap Values

This example shows how to set values in an `ESMF_BundleDataMap`.

```
il = ESMF_INTERLEAVE_BY_ITEM
call ESMF_BundleDataMapSet(bundleDM, bundleInterleave=il, rc=rc)


print *, "BundleDataMap after setting interleave = "
call ESMF_BundleDataMapPrint(bundleDM, rc=rc)
```

19.3.3 Getting BundleDataMap Values

This example shows how to query an `ESMF_BundleDataMap`.

```
call ESMF_BundleDataMapGet(bundleDM, bundleInterleave = il, rc=rc)
if (il .eq. ESMF_INTERLEAVE_BY_ITEM) then
  print *, "Interleaved by individual data items"
else if (il .eq. ESMF_INTERLEAVE_BY_BLOCK) then
  print *, "Interleaved by fields"
endif


! finalize ESMF framework
call ESMF_Finalize(rc)


end program ESMF_BundleDataMapEx
```

19.4 Restrictions and Future Work

1. **BundleDataMap is a placeholder until packed Bundles are implemented.** Packed Bundles have not been implemented in this version of ESMF.

19.5 Class API

19.5.1 ESMF_BundleDataMapGet - Get values from a BundleDataMap

INTERFACE:

```
subroutine ESMF_BundleDataMapGet(bundledatamap, bundleinterleave, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap), intent(in) :: bundledatamap
type(ESMF_InterleaveFlag), intent(out), optional :: bundleinterleave
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return values from an `ESMF_BundleDataMap`.
The arguments are:

bundledatamap An ESMF_BundleDataMap.

[bundleinterleave] Type of interleave for ESMF_Bundle data if packed into a single array. Possible values are ESMF_INTERLEAVE_BY_ITEM and ESMF_INTERLEAVE_BY_FIELD.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

19.5.2 ESMF_BundleDataMapPrint - Print information about a BundleDataMap

INTERFACE:

```
subroutine ESMF_BundleDataMapPrint(bundledatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap), intent(in) :: bundledatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints diagnostic information about the bundledatamap to stdout.
The arguments are:

bundledatamap ESMF_BundleDataMap to print.

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

19.5.3 ESMF_BundleDataMapSet - Set values in a BundleDataMap

INTERFACE:

```
subroutine ESMF_BundleDataMapSet(bundledatamap, bundleinterleave, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap), intent(inout) :: bundledatamap
type(ESMF_InterleaveFlag), intent(in), optional :: bundleinterleave
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set values in an ESMF_BundleDataMap.
The arguments are:

bundledatamap An ESMF_BundleDataMap.

[bundleinterleave] Type of interleave for ESMF_Bundle data if packed into a single array. Options are ESMF_INTERLEAVE_BY_IT and ESMF_INTERLEAVE_BY_FIELD.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

19.5.4 ESMF_BundleDataMapSetDefault - Set BundleDataMap default values

INTERFACE:

```
subroutine ESMF_BundleDataMapSetDefault(bundledatamap, bundleinterleave, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap) :: bundledatamap
type(ESMF_InterleaveFlag), intent(in), optional :: bundleinterleave
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set default values of a `ESMF_BundleDataMap` type. The differences between this routine and `ESMF_BundleDataMapSet()` is that unspecified arguments are reset to their default values.
The arguments are:

bundledatamap An `ESMF_BundleDataMap`.

[bundleinterleave] Type of interleave for `ESMF_Bundle` data if packed into a single array. Options are `ESMF_INTERLEAVE_BY_IT` and `ESMF_INTERLEAVE_BY_FIELD`. If not specified, the default is interleave by field.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

19.5.5 ESMF_BundleDataMapSetInvalid - Set BundleDataMap to invalid status

INTERFACE:

```
subroutine ESMF_BundleDataMapSetInvalid(bundledatamap, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap), intent(inout) :: bundledatamap
integer, intent(out), optional :: rc
```

DESCRIPTION:

ESMF routine to set the contents of an `ESMF_BundleDataMap` to an invalid status.
The arguments are:

bundledatamap An `ESMF_BundleDataMap`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

19.5.6 ESMF_BundleDataMapValidate - Check validity of a BundleDataMap

INTERFACE:

```
subroutine ESMF_BundleDataMapValidate(bundledatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_BundleDataMap), intent(in) :: bundledatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `bundledatamap` is internally consistent. Currently this method determines if the `bundledatamap` is uninitialized or already destroyed. The method returns an error code if problems are found.
The arguments are:

bundledatamap `ESMF_BundleDataMap` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if the `bundledatamap` is valid.

20 Field Class

20.1 Description

A Field represents a scalar physical field, such as temperature. ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects. The ESMF Field class contains the discretized field data, a reference to its associated grid, and metadata.
The Field class maintains the relationship of how data maps onto the Grid (e.g. one item per cell located at the cell center, one item per cell located at the NW corner, one item per cell vertex, etc). This means that different Fields which are on the same underlying Grid but have different mappings (staggerings) can share the same Grid object without needing to copy or replicate it multiple times.
The Field class provides methods for initialization, setting and retrieving data values, I/O, general data redistribution and regridding, standard communication methods such as gather and scatter, and manipulation of attributes.

20.2 Use and Examples

A Field serves as an annotator of data, since it carries a description of the grid it is associated with and metadata such as name and units. Fields can be used in this capacity alone, as convenient, descriptive containers into which arrays can be placed and retrieved. However, for most codes the primary use of Fields is in the context of import and export States, which are the objects that carry coupling information between Components. Fields enable data to be self-describing, and a State holding ESMF Fields contains data in a standard format that is easy to query and manipulate.
The information that is necessary for describing a Field to another Component is similar to the information needed to write history files. Another use of Fields is as a mechanism for writing out data to files for history and restart.
The sections below go into more detail about Field usage.

20.2.1 Field Creation

Fields can be created and destroyed at any time during application execution. However, these Field methods require some time to complete. We do not recommend that the user create or destroy Fields inside performance-critical computational loops.
All versions of the `ESMF_FieldCreate()` routines require a Grid object as input, or require a Grid be added before most operations involving Fields can be performed. The Grid contains the information needed to know which Decomposition Elements (DEs) are participating in the processing of this Field, and which subsets of the data are local to a particular DE.
The details of how the create process happens depends on which of the variants of the `ESMF_FieldCreate()` call is used. Some of the variants are discussed below.
There are versions of the `ESMF_FieldCreate()` interface which create the Field based on the input Grid. The ESMF can allocate the proper amount of space but not assign initial values. The user code can then get the pointer to the uninitialized buffer and set the initial data values.
Other versions of the `ESMF_FieldCreate()` interface allow user code to attach arrays that have already been allocated by the user. Empty Fields can also be created in which case the data can be added at some later time.

For versions of Create which do not specify data values, user code can create an ArraySpec object, which contains information about the Type, Kind, and Rank of the data values in the array. Then at Field create time, the appropriate amount of memory is allocated to contain the data which is local to each DE.

20.2.2 Field Deletion

There is a `ESMF_FieldDestroy()` method which releases any data buffers which were allocated or copied by this Field, and deletes the Field object. Since a single Grid reference can be shared by multiple Fields, the Grid is not deleted by this call.

```
! !PROGRAM: ESMF_FieldCreateEx - Field creation
!
! !DESCRIPTION:
!
! This program shows examples of Field initialization and manipulation
!-----------------------------------------------------------------------------

    ! ESMF Framework module
    use ESMF_Mod
    implicit none

    ! Local variables
    integer :: x, y, rc, mycell
    type(ESMF_Grid) :: grid
    type(ESMF_ArraySpec) :: arrayspec
    type(ESMF_Array) :: array1, array2
    type(ESMF_FieldDataMap) :: datamap
    type(ESMF_DELayout) :: layout
    type(ESMF_VM) :: vm
    type(ESMF_RelLoc) :: relativelocation
    character (len = ESMF_MAXSTR) :: fname
    type(ESMF_IOSpec) :: iospec
    type(ESMF_Field) :: field1, field2, field3
    real (ESMF_KIND_R8), dimension(:,:), pointer :: f90ptr1, f90ptr2
    real (ESMF_KIND_R8), dimension(2) :: origin
```

20.2.3 Field Create with Grid and Array

The user has already created an `ESMF_Grid` and an `ESMF_Array` with data. This create associates the two objects. An `ESMF_FieldDataMap` is created with all defaults.

```
    field1 = ESMF_FieldCreate(grid, array1, &
                         horzRelloc=ESMF_CELL_CENTER, name="pressure", rc=rc)
```

20.2.4 Field Create with Grid and ArraySpec

The user has already created an `ESMF_Grid` and an `ESMF_ArraySpec` which describes the data. This version of create will create an `ESMF_Array` based on the grid size and the `ESMF_ArraySpec`. An `ESMF_FieldDataMap` is created with all defaults.

```
    call ESMF_ArraySpecSet(arrayspec, 2, ESMF_DATA_REAL, ESMF_R4, rc)


    field2 = ESMF_FieldCreate(grid, arrayspec, horzRelloc=ESMF_CELL_CENTER, &
                              name="rh", rc=rc)
```

20.2.5 Empty Field Create

The user creates an empty ESMF_Field object. The ESMF_Grid, ESMF_Array, and ESMF_FieldDataMap can be added later using the set methods.

```
field3 = ESMF_FieldCreateNoData("precip", rc=rc)
```

20.2.6 Destroy a Field

When finished with an ESMF_Field, the destroy method removes it. However, the objects inside the ESMF_Field should be deleted separately, since objects can be added to more than one ESMF_Field, for example the same ESMF_Grid can be used in multiple ESMF_Fields.

```
call ESMF_FieldDestroy(field1, rc=rc)


 end program ESMF_FieldCreateEx
```

20.3 Restrictions and Future Work

1. **No mathematical operators.** The Fields class does not currently support advanced operations on fields, such as differential or other mathematical operators.
2. **No vector Fields.** ESMF does not currently support storage of multiple vector Field components in the same Field component, although that support is planned. At this time users need to create a separate Field object to represent each vector component.

20.4 Design and Implementation Notes

1. The Field class aggregates two internal subclasses: a GlobalField class and a LocalField class. This separation allows the code to clearly differentiate between functions which operate internal to a single DE on a local decomposition of data, and those which must be aware of the global state of the Field. There is a correspondence between the Global Distributed Grid class, and the Local Distributed Grid class, and Fields. Each DE contains the corresponding local decompositions for Distributed Grids and Fields.
2. Some methods which have a Field interface will actually be implemented at the underlying Grid or Array level; they will be inherited by the Field class. This allows the user API (Application Programming Interface) to present functions at the level which is most consistent to the application without restricting where inside the ESMF the actual implementation is done.

20.5 Class API: Basic Field Methods

20.5.1 ESMF_FieldCreateNoData - Create a Field with no associated data buffer

INTERFACE:

```
! Private name; call using ESMF_FieldCreateNoData()
function ESMF_FieldCreateNoDataPtr(grid, arrayspec, horzRelloc, vertRelloc, &
                                   haloWidth, datamap, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateNoDataPtr
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_ArraySpec), intent(in) :: arrayspec
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to `ESMF_FieldCreateNoData()`. Creates an `ESMF_Field` in its entirety except for the assignment or allocation of an associated raw data buffer.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

arrayspec Data specification.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.2 ESMF_FieldCreateNoData - Create a Field with no associated Array object

INTERFACE:

```
! Private name; call using ESMF_FieldCreateNoData()
function ESMF_FieldCreateNoArray(grid, horzRelloc, vertRelloc, &
                                 haloWidth, datamap, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateNoArray
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to `ESMF_FieldCreateNoData()`. This version of `ESMF_FieldCreate` builds an `ESMF_Field` and depends on a later call to add an `ESMF_Array` to it.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.3 ESMF_FieldCreateNoData - Create a Field with no Grid or Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreateNoData()
function ESMF_FieldCreateNoGridArray(name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateNoGridArray
```

ARGUMENTS:

```
character (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to `ESMF_FieldCreateNoData()`. This version of `ESMF_FieldCreate` builds an empty `ESMF_Field` and depends on later calls to add an `ESMF_Grid` and `ESMF_Array` to it.
The arguments are:

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.4 ESMF_FieldDestroy - Free all resources associated with a Field

INTERFACE:

```
subroutine ESMF_FieldDestroy(field, rc)
```

ARGUMENTS:

```
type(ESMF_Field) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with the `ESMF_Field`.
The arguments are:

field Pointer to an `ESMF_Field` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.5 ESMF_FieldGet - Return info associated with a Field

INTERFACE:

```
subroutine ESMF_FieldGet(field, grid, array, datamap, horzRelloc, &
                         vertRelloc, name, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
type(ESMF_Grid), intent(out), optional :: grid
type(ESMF_Array), intent(out), optional :: array
type(ESMF_FieldDataMap), intent(out), optional :: datamap
type(ESMF_RelLoc), intent(out), optional :: horzRelloc
type(ESMF_RelLoc), intent(out), optional :: vertRelloc
character(len=*), intent(out), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Query an `ESMF_Field` for various things. All arguments after the `Field` are optional. To select individual items use the named_argument=value syntax.
The arguments are:

ftype Pointer to an `ESMF_Field` object.

[grid] `ESMF_Grid`.

[array] `ESMF_Array`.

[datamap] `ESMF_FieldDataMap`.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[name] Name of queried item.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.6 ESMF_FieldGetArray - Get data Array associated with the Field

INTERFACE:

```
subroutine ESMF_FieldGetArray(field, array, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
type(ESMF_Array), intent(out) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Get data in `ESMF_Array` form.
The arguments are:

field An `ESMF_Field` object.

[array] Field `ESMF_Array`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.7 ESMF_FieldGetAttribute - Retrieve a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetInt4Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an integer attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.8 ESMF_FieldGetAttribute - Retrieve a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetInt4ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer list attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.9 ESMF_FieldGetAttribute - Retrieve a 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetInt8Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an integer attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.10 ESMF_FieldGetAttribute - Retrieve a 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetInt8ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte integer list attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.11 ESMF_FieldGetAttribute - Retrieve a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetReal4Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.12 ESMF_FieldGetAttribute - Retrieve a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetReal4ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from an `ESMF_Field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.13 ESMF_FieldGetAttribute - Retrieve a 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetReal8Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte real attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.14 ESMF_FieldGetAttribute - Retrieve a 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetReal8ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte real attribute from an `ESMF_Field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.15 ESMF_FieldGetAttribute - Retrieve a logical attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetLogicalAttr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The logical value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.16 ESMF_FieldGetAttribute - Retrieve a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetLogicalListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical list attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The logical values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.17 ESMF_FieldGetAttribute - Retrieve a character attribute

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttribute()
subroutine ESMF_FieldGetCharAttr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a character attribute from the `field`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to retrieve.

value The character value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.18 ESMF_FieldGetAttributeCount - Query the number of attributes

INTERFACE:

```
subroutine ESMF_FieldGetAttributeCount(field, count, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
integer, intent(out) :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the number of attributes associated with the given `field` in the argument `count`.
The arguments are:

field An `ESMF_Field` object.

count The number of attributes associated with this object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.19 ESMF_FieldGetAttributeInfo - Query Field attributes by name

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttributeInfo()
subroutine ESMF_FieldGetAttrInfoByName(field, name, datatype, &
                                        datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character(len=*), intent(in) :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the named attribute, including `datatype` and `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to query.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] The number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.20 ESMF_FieldGetAttributeInfo - Query Field attributes by index number

INTERFACE:

```
! Private name; call using ESMF_FieldGetAttributeInfo()
subroutine ESMF_FieldGetAttrInfoByNum(field, attributeIndex, name, &
                                       datatype, datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
integer, intent(in) :: attributeIndex
character(len=*), intent(out), optional :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed attribute, including `name`, `datatype`, `datakind` (if applicable) and `count`.
The arguments are:

field An `ESMF_Field` object.

attributeIndex The index number of the attribute to query.

name Returns the name of the attribute.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] Returns the number of items in this attribute. For character types, this is the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.21 ESMF_FieldPrint - Print the contents of a Field

INTERFACE:

```
subroutine ESMF_FieldPrint(field, options, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `field` to `stdout`.
The arguments are:

field

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.22 ESMF_FieldSetArray - Set data Array associated with the Field

INTERFACE:

```
subroutine ESMF_FieldSetArray(field, array, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Array), intent(in) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set data in ESMF_Array form.
The arguments are:

field An ESMF_Field object.

[array] ESMF_Array containing data.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.23 ESMF_FieldSetAttribute - Set a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetInt4Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer attribute to the field. The attribute has a name and a value.
The arguments are:

field An ESMF_Field object.

name The name of the attribute to add.

value The integer value of the attribute to add.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.24 ESMF_FieldSetAttribute - Set a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetInt4ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer list attribute to the `field`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.25 ESMF_FieldSetAttribute - Set a 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetInt8Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer attribute to the `field`. The attribute has a `name` and a `value`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

value The integer value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.26 ESMF_FieldSetAttribute - Set a 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetInt8ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer list attribute to the `field`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.27 ESMF_FieldSetAttribute - Set a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetReal4Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real attribute to the `field`. The attribute has a `name` and a `value`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

value The real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.28 ESMF_FieldSetAttribute - Set a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetReal4ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real list attribute to the `field`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.29 ESMF_FieldSetAttribute - Set a 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetReal8Attr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte real attribute to the `field`. The attribute has a `name` and a `value`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

value The real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.30 ESMF_FieldSetAttribute - Set a 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetReal8ListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte real list attribute to the `field`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.31 ESMF_FieldSetAttribute - Set a logical attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetLogicalAttr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical attribute to the `field`. The attribute has a `name` and a `value`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

value The logical true/false value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.32 ESMF_FieldSetAttribute - Set a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetLogicalListAttr(field, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical list attribute to the `field`. The attribute has a `name` and a `valueList`. The number of logical items in the `valueList` is given by `count`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

count The number of logicals in the `valueList`.

value The logical true/false values of the attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.33 ESMF_FieldSetAttribute - Set a character attribute

INTERFACE:

```
! Private name; call using ESMF_FieldSetAttribute()
subroutine ESMF_FieldSetCharAttr(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: name
character (len = *), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a character attribute to the `field`. The attribute has a `name` and a `value`.
The arguments are:

field An `ESMF_Field` object.

name The name of the attribute to add.

value The character value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.34 ESMF_FieldSetGrid - Set Grid associated with the Field

INTERFACE:

```
subroutine ESMF_FieldSetGrid(field, grid, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Grid), intent(in) :: grid
integer, intent(out), optional :: rc
```

DESCRIPTION:

Used only with the version of `ESMF_FieldCreate` which creates an empty `ESMF_Field` and allows the `ESMF_Grid` to be specified later. Otherwise it is an error to try to change the `ESMF_Grid` associated with an `ESMF_Field`.
The arguments are:

field An `ESMF_Field` object.

grid `ESMF_Grid` to be added.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.35 ESMF_FieldSetDataMap - Set DataMap assocated with a Field

INTERFACE:

```
subroutine ESMF_FieldSetDataMap(field, datamap, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_FieldDataMap), intent(in) :: datamap
integer, intent(out), optional :: rc
```

DESCRIPTION:

Used to set the ordering of an `ESMF_Field`. If an initialized `ESMF_FieldDataMap` and associated data are already in the `ESMF_Field`, the data will be reordered according to the new specification.
The arguments are:

field An `ESMF_Field` object.

datamap New memory order of data.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.5.36 ESMF_FieldValidate - Check validity of a Field

INTERFACE:

```
subroutine ESMF_FieldValidate(field, options, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
character (len = *), intent(in) :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `field` is internally consistent. Currently this method determines if the `field` is uninitialized or already destroyed. The method returns an error code if problems are found.
The arguments are:

field `ESMF_Field` to validate.

options Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if the `field` is valid.

20.5.37 ESMF_FieldWrite - Write a Field to external storage

INTERFACE:

```
subroutine ESMF_FieldWrite(field, iospec, timestamp, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
type(ESMF_IOSpec), intent(in), optional :: iospec
type(ESMF_Time), intent(in), optional :: timestamp
integer, intent(out), optional :: rc               ! return code
```

DESCRIPTION:

Used to write data to persistent storage in a variety of formats. (see WriteRestart/ReadRestart for quick data dumps.) Details of I/O options specified in the IOSpec derived type.
The arguments are:

name An `ESMF_Field` name.

[iospec] I/O specification.

[timestamp] `ESMF_Time`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6 Class API: Field Overloads for Fortran Arrays

20.6.1 ESMF_FieldCreate - Create a new Field

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateNew(grid, arrayspec, allocflag, horzRelloc, &
                             vertRelloc, haloWidth, datamap, name, &
                             iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateNew
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_ArraySpec), intent(in) :: arrayspec
type(ESMF_AllocFlag), intent(in), optional :: allocflag
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to `ESMF_FieldCreate()`. Create an `ESMF_Field` and allocate space internally for a gridded `ESMF_Array`. Return a new `ESMF_Field`.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

arrayspec `ESMF_Data` specification.

[allocflag] Whether to allocate space for the array. See Section 10.1.1 for possible values. Default is `ESMF_ALLOC`.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6.2 ESMF_FieldCreate - Create a Field from an existing ESMF Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateFromArray(grid, array, copyflag, horzRelloc, &
                                   vertRelloc, haloWidth, datamap, name, &
                                   iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateFromArray
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_Array), intent(in) :: array
type(ESMF_CopyFlag), intent(in), optional :: copyflag
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to `ESMF_FieldCreate()`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

array Includes data specification and allocated memory.

[copyflag] Indicates whether to reference the array or make a copy of it. Valid values are `ESMF_DATA_COPY` and `ESMF_DATA_REF`, respectively.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6.3 ESMF_FieldCreate - Create a Field by remapping another Field

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateRemap(srcField, grid, horzRelloc, vertRelloc, &
                               haloWidth, datamap, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateRemap
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: srcField
type(ESMF_Grid), intent(in) :: grid
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

An interface function to ESMF_FieldCreate(). Remaps data between an existing ESMF_Grid on a source ESMF_Field and a new ESMF_Grid. The ESMF_Grid is referenced by the new ESMF_Field. Data is copied.
The arguments are:

srcField Source ESMF_Field.

grid ESMF_Grid of source ESMF_Field.

horzRelLoc Relative location of data per grid cell/vertex in the horizontal grid.

vertRelLoc Relative location of data per grid cell/vertex in the vertical grid.

[halowidth] Halo width.

[datamap] ESMF_FieldDataMap

[name] ESMF_Field name.

[iospec] ESMF_Field ESMF_IOSpec.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.6.4 ESMF_FieldCreate - Create a Field using an existing Fortran data pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateDPtr<rank><type><kind>(grid, fptr, copyflag, &
horzRelloc, vertRelloc, haloWidth, datamap, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateDPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: fptr
type(ESMF_CopyFlag), intent(in) :: copyflag
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(in), optional :: haloWidth
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and associate the data in the Fortran array with the ESMF_Field. Return a new ESMF_Field.
Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

fptr A Fortran array pointer which must be already allocated and the proper size for this portion of the grid.

copyflag Whether to copy the existing data space or reference directly. Default is `ESMF_DATA_COPY`. Other option is `ESMF_DATA_REF`.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6.5 ESMF_FieldCreate - Create a Field using an unallocated Fortran data pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateEPtr<rank><type><kind>(grid, fptr, allocflag, &
horzRelloc, vertRelloc, haloWidth, lbounds, ubounds, &
datamap, name, iospec, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateEPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: fptr
type(ESMF_AllocFlag), intent(in), optional :: allocflag
integer, intent(in), optional :: haloWidth
integer, dimension(:), intent(in), optional :: lbounds
integer, dimension(:), intent(in), optional :: ubounds
type(ESMF_FieldDataMap), intent(in), optional :: datamap
character (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`, allocate necessary data space, and return with the Fortran array pointer initialized to point to the data space. Function return value is the new `ESMF_Field`. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

grid Pointer to an `ESMF_Grid` object.

fptr A Fortran array pointer which must be unallocated but of the proper rank, type, and kind for the data to be associated with this `EWSF_Field`.

[allocflag] Whether to allocate space for the array. See Section 10.1.1 for possible values. Default is `ESMF_ALLOC`.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[haloWidth] Maximum halo depth along all edges. Default is 0.

[lbounds] An integer array of lower index values. Must be the same length as the rank.

[ubounds] An integer array of upper index values. Must be the same length as the rank.

[datamap] Describes the mapping of data to the `ESMF_Grid`.

[name] `Field` name.

[iospec] I/O specification.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6.6 ESMF_FieldGetDataPointer - Retrieve Fortran pointer directly from a Field

INTERFACE:

```
! Private name; call using ESMF_FieldGetDataPointer()
subroutine ESMF_FieldGetDataPointer<rank><type><kind>(field, ptr, copyflag, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: ptr
type(ESMF_CopyFlag), intent(in), optional :: copyflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a direct Fortran pointer to the data in an `ESMF_Field`. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

field The `ESMF_Field` to query.

ptr An unassociated Fortran pointer of the proper Type, Kind, and Rank as the data in the Field. When this call returns successfully, the pointer will now point to the data in the Field. This is either a reference or a copy, depending on the setting of the following argument.

[copyflag] Defaults to `ESMF_DATA_REF`. If set to `ESMF_DATA_COPY`, a separate copy of the data will be allocated and the pointer will point at the copy. If a new copy of the data is made, the caller is responsible for deallocating the space.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.6.7 ESMF_FieldSetDataPointer - Add data to a field directly by Fortran pointer

INTERFACE:

```
! Private name; call using ESMF_FieldSetDataPointer()
subroutine ESMF_FieldSetDataPointer<rank><type><kind>(field, &
dataPointer, copyflag, indexflag, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: dataPointer
integer, intent(in), optional :: haloWidth
type(ESMF_CopyFlag), intent(in), optional :: copyflag
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set data in an ESMF_Field directly from a Fortran pointer. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

field The ESMF_Field to query.

dataPointer An associated Fortran pointer of the proper Type, Kind, and Rank as the data in the Field. When this call returns successfully, the pointer will now point to the data in the Field. This is either a reference or a copy, depending on the setting of the following argument.

[copyflag] Defaults to ESMF_DATA_REF. If set to ESMF_DATA_COPY, a separate copy of the data will be allocated and the pointer will point at the copy. If a new copy of the data is made, the caller is responsible for deallocating the space.

[haloWidth] Defaults to 0. If specified, the halo width to add to all sides of the data array.

[indexflag] See Section 10.1.4 for possible values. Defaults to ESMF_INDEX_DELOCAL. If set to ESMF_INDEX_GLOBAL and the ESMF_Grid associated with the ESMF_Field is regular, then the lower bounds and upper bounds will be allocated with global index numbers corresponding to the grid.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7 Class API: Field Communications

20.7.1 ESMF_FieldAllGather - Data allgather operation on a Field

INTERFACE:

```
subroutine ESMF_FieldAllGather(field, array, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Array), intent(out) :: array
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
type(ESMF_CommHandle), intent(inout), optional :: commhandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform an allgather operation over the data in an ESMF_Field. If the ESMF_Field is decomposed over N DEs, this routine returns a copy of the entire collected data ESMF_Array on each of the N DEs.
The arguments are:

field ESMF_Field containing data to be gathered.

array Newly created array containing the collected data. It is the size of the entire undecomposed grid.

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communications. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blockingflag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.2 ESMF_FieldGather - Data gather operation on a Field

INTERFACE:

```
subroutine ESMF_FieldGather(field, dstDe, array, blockingflag, &
                            commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
integer, intent(in) :: dstDe
type(ESMF_Array), intent(out) :: array
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
type(ESMF_CommHandle), intent(inout), optional :: commhandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a gather operation over the data in an ESMF_Field. If the ESMF_Field is decomposed over N DEs, this routine returns a copy of the entire collected data as an ESMF_Array on the specified destination DE number. On all other DEs there is no return array value.
The arguments are:

field ESMF_Field containing data to be gathered.

dstDe Destination DE number where the gathered data is to be returned.

array Newly created ESMF_Array containing the collected data on the specified DE. It is the size of the entire undecomposed grid. On all other DEs this argument returns an invalid object.

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, the default is to do synchronous communications. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blockingflag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.3 ESMF_FieldHalo - Execute a halo operation on a Field

INTERFACE:

```
! Private name; call using ESMF_FieldHalo()
subroutine ESMF_FieldHaloRun(field, routehandle, blockingflag, &
                             commhandle, halodirection, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
type(ESMF_CommHandle), intent(inout), optional :: commhandle
type(ESMF_HaloDirection), intent(in), optional :: halodirection
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a halo operation over the data in an `ESMF_Field`. This routine updates the data inside the `ESMF_Field` in place.
The arguments are:

field `ESMF_Field` containing data to be haloed.

routehandle `ESMF_RouteHandle` which was returned by the corresponding `ESMF_FieldHaloStore()` call. It is associated with the precomputed data movement and communication needed to perform the halo operation.

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is what was specified at Store time. If `both` was specified at Store time, this defaults to blocking. Valid values for this flag are `ESMF_BLOCKING` and `ESMF_NONBLOCKING`. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the `blockingflag` is set to `ESMF_NONBLOCKING` this argument is required. Information about the pending operation will be stored in the `ESMF_CommHandle` and can be queried or waited for later.

[halodirection] Optional argument to restrict halo direction to a subset of the possible halo directions. If not specified, the halo is executed along all boundaries. This option is used only in the situation where the halo must be precomputed at this time. (This feature is not yet supported.)

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.7.4 ESMF_FieldHaloRelease - Release resources associated w/ handle

INTERFACE:

```
subroutine ESMF_FieldHaloRelease(routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Release all stored information about the halo operation associated with this `ESMF_RouteHandle`.
The arguments are:

routehandle ESMF_RouteHandle associated with this halo operation.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.5 ESMF_FieldHaloStore - Precompute a halo operation on a Field

INTERFACE:

```
subroutine ESMF_FieldHaloStore(field, routehandle, halodirection, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_HaloDirection), intent(in), optional :: halodirection
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute the data movement or communication operations needed to perform a halo operation over the data in an ESMF_Field. The list of operations will be associated internally to the framework with the ESMF_RouteHandle object. To perform the actual halo operation the ESMF_FieldHalo() routine must be called with the ESMF_Field containing the data to be updated and the ESMF_RouteHandle computed during this store call. If more than one ESMF_Field has identical ESMF_Grids and ESMF_FieldDataMaps, then the same ESMF_RouteHandle can be computed once and used in multiple executions of the halo operation.
The arguments are:

field ESMF_Field containing data to be haloed.

routehandle ESMF_RouteHandle which will be returned after being associated with the precomputed information for a halo operation on this ESMF_Field. This handle must be supplied at run time to execute the halo.

[halodirection] Optional argument to restrict halo direction to a subset of the possible halo directions. If not specified, the halo is executed along all boundaries. (This feature is not yet supported.)

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.6 ESMF_FieldRedist - Data redistribution operation on a Field

INTERFACE:

```
subroutine ESMF_FieldRedist(srcField, dstField, routehandle, blockingflag, &
                            commhandle, parentDelayout, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: srcField
type(ESMF_Field), intent(inout) :: dstField
type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
type(ESMF_CommHandle), intent(inout), optional :: commhandle
type(ESMF_DELayout), intent(in), optional :: parentDelayout
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a redistribution operation over the data in an ESMF_Field. This routine reads the source field and leaves the data untouched. It reads the ESMF_Grid and ESMF_FieldDataMap from the destination field and updates the array data in the destination. The ESMF_Grids may have different decompositions (different ESMF_DELayouts) or different data maps, but the source and destination grids must describe the same set of coordinates. Unlike ESMF_FieldRegrid this routine does not do interpolation, only data movement.
The arguments are:

srcField ESMF_Field containing source data.

dstField ESMF_Field containing destination grid.

routehandle ESMF_RouteHandle which was returned by the corresponding ESMF_FieldRedistStore() call. It is associated with the precomputed data movement and communication needed to perform the redistribution operation.

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communication. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blockingflag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[parentDelayout] ESMF_DELayout which encompasses both ESMF_Fields, most commonly the layout of the Coupler if the redistribution is inter-component, but could also be the individual layout for a component if the redistribution is intra-component. This argument is only used in the situation where the routehandle has not been precomputed yet.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.7 ESMF_FieldRedistRelease - Release resources associated w/ handle

INTERFACE:

```
subroutine ESMF_FieldRedistRelease(routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Release all stored information about the redistribution associated with this ESMF_RouteHandle.
The arguments are:

routehandle ESMF_RouteHandle associated with this redistribution.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.8 ESMF_FieldRedistStore - Data redistribution operation on a Field

INTERFACE:

```
subroutine ESMF_FieldRedistStore(srcField, dstField, parentDelayout, &
                                 routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: srcField
type(ESMF_Field), intent(inout) :: dstField
type(ESMF_DELayout), intent(in) :: parentDelayout
type(ESMF_RouteHandle), intent(out) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute the data movement or communications operations needed to accomplish a data redistribution operation over the data in an `ESMF_Field`. Data redistribution differs from regridding in that redistribution does no interpolation, only a 1-for-1 movement of data from one location to another. Therefore, while the `ESMF_Grid`s for the source and destination may have different decompositions (different `ESMF_DELayout`s) or different data maps, the source and destination grids must describe the same set of coordinates.
The arguments are:

srcField `ESMF_Field` containing source data.

dstField `ESMF_Field` containing destination grid.

parentDelayout `ESMF_DELayout` which encompasses both `ESMF_Field`s, most commonly the layout of the Coupler if the redistribution is inter-component, but could also be the individual layout for a component if the redistribution is intra-component.

routehandle `ESMF_RouteHandle` which will be used to execute the redistribution when `ESMF_FieldRedist` is called.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.7.9 ESMF_FieldRedistStore - Data redistribution operation on a Field

INTERFACE:

```
subroutine ESMF_FieldRedistStoreNew(srcField, decompIds, dstField, &
                                    parentDelayout, routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: srcField
integer, dimension(:), intent(in) :: decompIds
type(ESMF_Field), intent(out) :: dstField
type(ESMF_DELayout), intent(in) :: parentDelayout
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute a redistribution operation over the data in a `ESMF_Field`. This routine reads the source field and leaves the data untouched. This version of RedistStore creates the destination `ESMF_Field` and its underlying `ESMF_Grid` and `ESMF_FieldDataMap` from the source grid and input decompIds. Unlike `ESMF_FieldRegrid` this routine does not do interpolation, only data movement.
The arguments are:

srcField ESMF_Field containing source data.

decompIds Array of decomposition identifiers.

dstField ESMF_Field containing destination grid.

parentDelayout ESMF_DELayout which encompasses both ESMF_Fields, most commonly the layout of the Coupler if the redistribution is inter-component, but could also be the individual layout for a component if the redistribution is intra-component.

routehandle ESMF_RouteHandle which will be used to execute the redistribution when ESMF_FieldRedist is called.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.10 ESMF_FieldRegrid - Data regrid operation on a Field

INTERFACE:

```
      subroutine ESMF_FieldRegrid(srcField, dstField, routehandle, &
                                  parentDelayout, regridmethod, regridnorm, &
                                  srcMask, dstMask, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_Field), intent(in) :: srcField
      type(ESMF_Field), intent(inout) :: dstField
      type(ESMF_RouteHandle), intent(inout) :: routehandle
      type(ESMF_DELayout), intent(in), optional :: parentDelayout
      type(ESMF_RegridMethod), intent(in), optional :: regridmethod
      type(ESMF_RegridNormOpt), intent(in), optional :: regridnorm
      type(ESMF_Mask), intent(in), optional :: srcMask
      type(ESMF_Mask), intent(in), optional :: dstMask
      type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
      type(ESMF_CommHandle), intent(inout), optional :: commhandle
      integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a regrid operation over the data in an ESMF_Field. This routine reads the source field and leaves the data untouched. It uses the ESMF_Grid and ESMF_FieldDataMap information in the destination field to control the transformation of data. The array data in the destination field is overwritten by this call.
The arguments are:

srcField ESMF_Field containing source data.

dstField ESMF_Field containing destination grid and data map.

routehandle ESMF_RouteHandle which will be returned after being associated with the precomputed information for a regrid operation on this ESMF_Field. This handle must be supplied at run time to execute the regrid.

[parentDelayout] ESMF_DELayout which encompasses both ESMF_Fields, most commonly the layout of the Coupler if the regridding is inter-component, but could also be the individual layout for a component if the regridding is intra-component. This argument is used only if the routehandle has not been previously computed during a RegridStore call.

[regridmethod] Type of regridding to do. A set of predefined methods are supplied. This argument is used only if the routehandle has not been previously computed during a RegridStore call.

[regridnorm] Normalization option, only for specific regrid types. This argument is used only if the routehandle has not been previously computed during a RegridStore call.

[srcMask] Optional `ESMF_Mask` identifying valid source data. (Not yet implemented.)

[dstMask] Optional `ESMF_Mask` identifying valid destination data. (Not yet implemented.)

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communications. Valid values for this flag are `ESMF_BLOCKING` and `ESMF_NONBLOCKING`. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the `blockingflag` is set to `ESMF_NONBLOCKING` this argument is required. Information about the pending operation will be stored in the `ESMF_CommHandle` and can be queried or waited for later.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.7.11 ESMF_FieldRegridRelease - Release information for this handle

INTERFACE:

```
subroutine ESMF_FieldRegridRelease(routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Release all stored information about the regridding associated with this `ESMF_RouteHandle`.
The arguments are:

routehandle `ESMF_RouteHandle` associated with this regrid operation.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

20.7.12 ESMF_FieldRegridStore - Data regrid operation on a Field

INTERFACE:

```
subroutine ESMF_FieldRegridStore(srcField, dstField, parentDelayout, &
                                  routehandle, regridmethod, regridnorm, &
                                  srcMask, dstMask, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: srcField
type(ESMF_Field), intent(inout) :: dstField
type(ESMF_DELayout), intent(in) :: parentDelayout
type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_RegridMethod), intent(in) :: regridmethod
type(ESMF_RegridNormOpt), intent(in), optional :: regridnorm
type(ESMF_Mask), intent(in), optional :: srcMask
type(ESMF_Mask), intent(in), optional :: dstMask
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute the data movement or communications operations plus the interpolation information needed to execute a regrid operation which will move and transform data from the source field to the destination field. This information is associated with the ESMF_RouteHandle which must then be supplied during the actual execution of the regrid operation.
The arguments are:

srcField ESMF_Field containing source data.

dstField ESMF_Field containing destination grid and data map.

parentDelayout ESMF_DELayout which encompasses both ESMF_Fields, most commonly the layout of the Coupler if the regridding is inter-component, but could also be the individual layout for a component if the regridding is intra-component.

routehandle Output from this call, identifies the precomputed work which will be executed when ESMF_FieldRegrid is called.

regridmethod Type of regridding to do. A set of predefined methods are supplied.

[regridnorm] Normalization option, only for specific regrid types.

[srcMask] Optional ESMF_Mask identifying valid source data.

[dstMask] Optional ESMF_Mask identifying valid destination data.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.7.13 ESMF_FieldScatter - Data scatter operation on a Field

INTERFACE:

```
subroutine ESMF_FieldScatter(array, srcDe, field, &
                             blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
integer, intent(in) :: srcDe
type(ESMF_Field), intent(inout) :: field
type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
type(ESMF_CommHandle), intent(inout), optional :: commhandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a scatter operation over the data in an ESMF_Array, returning it as the data array in an ESMF_Field. If the Field is decomposed over N DEs, this routine takes a single array on the specified DE and returns a decomposed copy on each of the N DEs, as the ESMF_Array associated with the given empty ESMF_Field.
The arguments are:

array Input ESMF_Array containing the collected data. It must be the size of the entire undecomposed grid.

srcDe Integer DE number where the data to be Scattered is located. The ESMF_Array input is ignored on all other DEs.

field Empty Field containing ESMF_Grid which will correspond to the data in the array which will be scattered. When this routine returns each ESMF_Field will contain a valid data array containing the subset of the decomposed data.

[blockingflag] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communications. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blockingflag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

21 FieldDataMap Class

21.1 Description

The FieldDataMap class describes how vector fields are interleaved. Since the ESMF Field class does not yet fully support vector fields, this class is simply a placeholder.

21.2 Use and Examples

FieldDataMaps are shallow objects. They can be declared as local (stack) variables in subroutines. They do not need a create or destroy method. There is a method to set the initial default values, to set and query individual values, and to print the contents in human-readable form for output or debugging.

```
! !PROGRAM: ESMF_FieldDataMapEx - Field DataMap manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Field DataMap set and get usage
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! local variables
      type(ESMF_FieldDataMap) :: fieldDM
      type(ESMF_IndexOrder) :: indexOrder
      type(ESMF_RelLoc) :: relativeLocation
      integer :: dataRank, dataIndexList(ESMF_MAXDIM), counts(ESMF_MAXDIM)



      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(rc=rc)
```

21.2.1 Setting Field DataMap Defaults and Invalidation

This example shows how to set the default values in an ESMF_FieldDataMap, and how to intentionally mark an ESMF_FieldDataMap invalid.

```
! Set up a default data map for a Field with 2D data,
! and a 1-for-1 mapping with the Grid.
call ESMF_FieldDataMapSetDefault(fieldDM, 2, rc=rc)


print *, "Default values for FieldDataMap = "
call ESMF_FieldDataMapPrint(fieldDM, rc=rc)


relativeLocation = ESMF_CELL_NECORNER
call ESMF_FieldDataMapSetDefault(fieldDM, ESMF_INDEX_IJK, &
                                 horzRelloc=relativeLocation, rc=rc)


print *, "FieldDataMap after set = "
call ESMF_FieldDataMapPrint(fieldDM, rc=rc)


call ESMF_FieldDataMapSetInvalid(fieldDM, rc=rc)


print *, "Invalid FieldDataMap = "
call ESMF_FieldDataMapPrint(fieldDM, rc=rc)
```

21.2.2 Setting Field DataMap Values

This example shows how to set values in an ESMF_FieldDataMap.

```
relativeLocation = ESMF_CELL_CENTER
call ESMF_FieldDataMapSet(fieldDM, dataRank=2, &
                          horzRelloc=relativeLocation, rc=rc)


print *, "FieldDataMap after set = "
call ESMF_FieldDataMapPrint(fieldDM, rc=rc)
```

21.2.3 Getting Field DataMap Values

This example shows how to query an ESMF_FieldDataMap.

```
call ESMF_FieldDataMapGet(fieldDM, dataRank, dataIndexList, &
                          horzRelloc=relativeLocation, rc=rc)
print *, "Returned values from Field DataMap:"
print *, "data rank: ", dataRank
print *, "mapping of grid to data indices: ", dataIndexList
```

```
! finalize ESMF framework
call ESMF_Finalize(rc)

end program ESMF_FieldDataMapEx
```

21.3 Restrictions and Future Work

1. **No support for vector Fields.** While vector interleave can be set and queried in a FieldDataMap, no support for it exists in other parts of this version of ESMF, since the Field class does not support vector Fields yet. The user can create a separate Field for each vector component.

21.4 Design and Implementation Notes

The FieldDataMap contains information needed by other objects in order to correctly handle data and grid operations. It is implemented as a simple Fortran derived type, and contains an ArrayDataMap object as well as additional information needed at the Field level.

21.5 Class API

21.5.1 ESMF_FieldDataMapGet - Get values from a FieldDataMap

INTERFACE:

```
subroutine ESMF_FieldDataMapGet(fielddatamap, dataRank, dataIndexList, counts, &
                                horzRelloc, vertRelloc, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap), intent(in) :: fielddatamap
integer, intent(out), optional :: dataRank
integer, dimension(:), intent(out), optional :: dataIndexList
integer, dimension(:), intent(out), optional :: counts
type(ESMF_RelLoc), intent(out), optional :: horzRelloc
type(ESMF_RelLoc), intent(out), optional :: vertRelloc
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return information about this `ESMF_FieldDataMap`.
The arguments are:

fielddatamap An `ESMF_FieldDataMap`.

[datarank] The number of dimensions in the data `ESMF_Array`.

[dataIndexList] An integer array, `datarank` long, which specifies the mapping between rank numbers in the `ESMF_Grid` and the `ESMF_Array`. If there is no correspondance (because the `ESMF_Array` has a higher rank than the `ESMF_Grid`) the index value will be 0.

[counts] An integer array, with length (`datarank` minus the grid rank). Each entry is the default item count which would be used for those ranks which do not correspond to grid ranks when creating an `ESMF_Field` using only an an `ESMF_ArraySpec` and an `ESMF_ArrayDataMap`.

[horzRelloc] Relative location of data per grid cell/vertex in the horzontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

21.5.2 ESMF_FieldDataMapPrint - Print a FieldDataMap

INTERFACE:

```
subroutine ESMF_FieldDataMapPrint(fielddatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap), intent(in) :: fielddatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `fielddatamap` to `stdout`.
The arguments are:

fielddatamap `ESMF_FieldDataMap` to print.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

21.5.3 ESMF_FieldDataMapSet - Set values in a FieldDataMap

INTERFACE:

```
subroutine ESMF_FieldDataMapSet(fielddatamap, dataRank, dataIndexList, counts, &
                                horzRelloc, vertRelloc, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap), intent(inout) :: fielddatamap
integer, intent(in), optional :: dataRank
integer, dimension(:), intent(in), optional :: dataIndexList
integer, dimension(:), intent(in), optional :: counts
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set values in an `ESMF_FieldDataMap`.
The arguments are:

fielddatamap An `ESMF_FieldDataMap`.

[datarank] The number of dimensions in the data `ESMF_Array`.

[dataIndexList] An integer array, `datarank` long, which specifies the mapping between rank numbers in the `ESMF_Grid` and the `ESMF_Array`. If there is no correspondance (because the `ESMF_Array` has a higher rank than the `ESMF_Grid`) the index value must be 0.

[counts] An integer array, with length (`datarank` minus the grid rank). If the `ESMF_Array` is a higher rank than the `ESMF_Grid`, the additional dimensions may optionally each have an item count defined here. This allows `ESMF_FieldCreate()` to take an `ESMF_ArraySpec` and an `ESMF_ArrayDataMap` and create the appropriately sized `ESMF_Array` for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if `ESMF_FieldCreate()` is called called with an already-created `ESMF_Array`.

[horzRelloc] Relative location of data per grid cell/vertex in the horzontal grid.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

21.5.4 ESMF_FieldDataMapSetDefault - Set FieldDataMap default values

INTERFACE:

```
subroutine ESMF_FieldDataMapSetDefExplicit(fielddatamap, dataRank, &
                                           dataIndexList, counts, &
                                           horzRelloc, vertRelloc, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap) :: fielddatamap
integer, intent(in) :: dataRank
integer, dimension(:), intent(in), optional :: dataIndexList
integer, dimension(:), intent(in), optional :: counts
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set default values of an ESMF_FieldDataMap. This differs from ESMF_FieldDataMapSet() in that all values which are not specified here will be overwritten with default values.

fielddatamap An ESMF_FieldDataMap.

datarank The number of dimensions in the data ESMF_Array.

[dataIndexList] An integer array, datarank long, which specifies the mapping between rank numbers in the ESMF_Grid and the ESMF_Array. If there is no correspondance (because the ESMF_Array has a higher rank than the ESMF_Grid) the index value must be 0. The default is a 1-to-1 mapping with the ESMF_Grid.

[counts] An integer array, with length (datarank minus the grid rank). If the ESMF_Array is a higher rank than the ESMF_Grid, the additional dimensions may optionally each have an item count defined here. This allows ESMF_FieldCreate() to take an ESMF_ArraySpec and an ESMF_ArrayDataMap and create the appropriately sized ESMF_Array for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if ESMF_FieldCreate() is called called with an already-created ESMF_Array. If unspecified, the default lengths are 1.

[horzRelloc] Relative location of data per grid cell/vertex in the horizontal grid. The default is ESMF_CELL_CENTER.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid. The default is ESMF_CELL_UNDEFINED.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

21.5.5 ESMF_FieldDataMapSetDefault - Set FieldDataMap default values

INTERFACE:

```
! Private name; call using ESMF_FieldDataMapSetDefault()
subroutine ESMF_FieldDataMapSetDefIndex(fielddatamap, indexorder, counts, &
                                        horzRelloc, vertRelloc, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap) :: fielddatamap
type(ESMF_IndexOrder), intent(in) :: indexorder
integer, dimension(:), intent(in), optional :: counts
type(ESMF_RelLoc), intent(in), optional :: horzRelloc
type(ESMF_RelLoc), intent(in), optional :: vertRelloc
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set default values of an `ESMF_FieldDataMap`. This differs from `ESMF_FieldDataMapSet()` in that all values which are not specified here will be overwritten with default values.

fielddatamap An `ESMF_FieldDataMap`.

indexorder An `ESMF_DataIndexOrder` which specifies one of several common predefined mappings between the grid and data ranks. This is simply a convenience for the common cases; there is a more general form of this call which allows the mapping to be specified as an integer array of index numbers directly.

[counts] An integer array, with length (`datarank` minus the grid rank). If the `ESMF_Array` is a higher rank than the `ESMF_Grid`, the additional dimensions may optionally each have an item count defined here. This allows `ESMF_FieldCreate()` to take an `ESMF_ArraySpec` and an `ESMF_ArrayDataMap` and create the appropriately sized `ESMF_Array` for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if `ESMF_FieldCreate()` is called called with an already-created `ESMF_Array`. If unspecified, the default lengths are 1.

[horzRelloc] Relative location of data per grid cell/vertex in the horzontal grid. The default is `ESMF_CELL_CENTER`.

[vertRelloc] Relative location of data per grid cell/vertex in the vertical grid. The default is `ESMF_CELL_UNDEFINED`.

21.5.6 ESMF_FieldDataMapSetInvalid - Set FieldDataMap to an invalid status

INTERFACE:

```
subroutine ESMF_FieldDataMapSetInvalid(fielddatamap, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap), intent(inout) :: fielddatamap
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set the contents of an `ESMF_FieldDataMap` to an invalid status.
The arguments are:

fielddatamap An `ESMF_FieldDataMap`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

21.5.7 ESMF_FieldDataMapValidate - Check validity of a FieldDataMap

INTERFACE:

```
subroutine ESMF_FieldDataMapValidate(fielddatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_FieldDataMap), intent(in) :: fielddatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `fielddatamap` is internally consistent. Currently this method determines if the `fielddatamap` is uninitialized or already destroyed. The method returns an error code if problems are found.
The arguments are:

fielddatamap `ESMF_FieldDataMap` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22 Array Class

22.1 Description

An ESMF Array, as one might expect, represents a multidimensional array. It can be real, integer, or logical, and can possess up to seven dimensions. The array can be strided. The first dimension specified is always the one which varies fastest in linearized memory.
Arrays can be created, destroyed, copied, and indexed. Communication methods, such as redistribution, are also defined. For information on Array halo domains and operations, see Section 28.1.

22.2 Use and Examples

The variants of Array create methods include 2 language interfaces, 3 allocation options, and 3 data type/kind/rank (TKR) specification options. From Fortran the create options are to specify TKR either explicitly, with an ArraySpec object, or to give a Fortran array pointer which is queried by the framework. The allocation options are to allocate uninitialized space, to allocate space and copy data values into it, or to reference already allocated data space. The `ESMF_ArrayCreate()` method is overloaded in Fortran with an interface block.
The Array get/set methods support returning either a pointer to the existing space, or allocating a new copy of the data and returning a pointer to the copy. The return value is a Fortran pointer to a specific TRK array. This allows standard Fortran array manipulations to be performed on the data without intervention of the framework.
The general allocation/deallocation rules are: if the ESMF allocated the space at create time then the Array destroy routine will deallocate the space at the time the Array object is deleted. If the user allocated the space by specifying the data reference option to the create method then the space will not be deallocated by the framework. The user is responsible for calling the corresponding language routine to return the space to the heap. If the user requests a copy of the data with a call to the `ESMF_ArrayGetData()` routine, they assume ownership of the copied buffer and are responsible for deallocating the space.

```
! !PROGRAM: ESMF_ArrayCreateEx - Examples of Array Creation
!
! !DESCRIPTION:
!
! This program shows examples of Array creation
```

```
! ESMF Framework module
use ESMF_Mod
implicit none

! Local variables
integer :: nx, ny, arank, rc
integer :: i, j, ni, nj
type(ESMF_ArraySpec) :: arrayspec
type(ESMF_DataKind) :: akind
type(ESMF_DataType) :: atype
type(ESMF_Array) :: array1, array2, array3
real(selected_real_kind(6,45)), dimension(:,:), pointer :: realptr, realptr2
integer(selected_int_kind(5)), dimension(:), pointer :: intptr, intptr2
```

22.2.1 Create an Array with Existing Data

Create an ESMF_Array based on an existing, allocated Fortran pointer. The data is type Integer, one dimensional. The ESMF_DATA_REF flag means the framework will not make a copy of the data area but will use this memory directly. When the ESMF_Array is deleted the data area will remain and must be deallocated by the user when the space is not needed.

```
! Allocate and set initial data values
ni = 15
allocate(intptr(ni))
do i=1,ni
   intptr(i) = i
enddo

array1 = ESMF_ArrayCreate(intptr, ESMF_DATA_REF, rc=rc)
```

22.2.2 Destroy an Array

When finished with an ESMF_Array, remove the object and release any resources associated with it.

```
call ESMF_ArrayDestroy(array1, rc)
```

22.2.3 Create an Array and Copy Existing Data

Create an ESMF_Array based on an existing, allocated Fortran pointer. The data is type Integer, one dimensional. The ESMF_DATA_COPY flag means the framework will make a copy of the data area and will be independent of the original data array. When the ESMF_Array is deleted this data area will be deallocated by the framework. The user can user or delete the original data area independently of this ESMF_Array.

```
! Allocate and set initial data values
ni = 5
nj = 3
allocate(realptr(ni,nj))
do i=1,ni
 do j=1,nj
   realptr(i,j) = i + ((j-1)*ni) + 0.1
 enddo
```

```
enddo

array2 = ESMF_ArrayCreate(realptr, ESMF_DATA_COPY, rc=rc)
```

22.2.4 Create an Array and Allocate Data Space

Create an ESMF_Array based on a description of the data. The framework will allocate the data space itself. When the ESMF_Array is deleted this data area will be deallocated by the framework.

```
arank = 2
atype = ESMF_DATA_REAL
akind = ESMF_R8

call ESMF_ArraySpecSet(arrayspec, arank, atype, akind)
array2 = ESMF_ArrayCreate(arrayspec, (/10, 20 /), rc=rc)


call ESMF_Finalize(rc)

end program ESMF_ArrayCreateEx
```

```
! !PROGRAM: ESMF_ArrayGetEx - Examples of Array Usage
!
! !DESCRIPTION:
!
! This program shows examples of Array creation
```

```
! ESMF Framework module
use ESMF_Mod
implicit none

! Local variables
integer :: nx, ny, arank, rc
integer :: i, j, ni, nj
type(ESMF_ArraySpec) :: arrayspec
type(ESMF_DataKind) :: akind
type(ESMF_DataType) :: atype
type(ESMF_Array) :: array1, array2, array3
real(selected_real_kind(6,45)), dimension(:,:), pointer :: realptr, realptr2
integer(selected_int_kind(5)), dimension(:), pointer :: intptr, intptr2
```

22.2.5 Print Array Contents

Print the data contents of an ESMF_Array.

```
call ESMF_ArrayPrint(array1, rc=rc)
```

22.2.6 Get a Pointer to the Array Contents

Associate a Fortran pointer with the data from an ESMF_Array. Point directly at the data contents; do not make a separate copy.

```
call ESMF_ArrayGetData(array1, intptr2, ESMF_DATA_REF, rc)
```

22.2.7 Destroy an Array

When finished with an ESMF_Array, remove the object and release any resources associated with it.

```
call ESMF_ArrayDestroy(array1, rc)
```

22.2.8 Get a Pointer to a Copy of the Array Contents

Associate a Fortran pointer with the data from an ESMF_Array. Allocate and copy the existing data into a separate buffer and return that space. It can be manipulated independently from the ESMF_Array contents. It must be deallocated by the user when no longer needed.

```
call ESMF_ArrayGetData(array2, realptr2, ESMF_DATA_COPY, rc)


call ESMF_Finalize(rc)

end program ESMF_ArrayGetEx
```

22.3 Restrictions and Future Work

1. **7D limit.** Scalars and ranks up to 7D (the Fortran limit) are supported.
2. **Arrays of derived types not supported.** Arrays of derived types will not be directly supported by the framework. However, non-contiguous arrays which can be defined by (start/end/stride) triplets will be, so the user can construct a buffer by using a derived type and then define it as individual data arrays or a packed bundle array.

22.4 Design and Implementation Notes

1. **Class and directory hierarchy.** The LocalArray class is an internal class which is not visible outside the framework. As described below it contains all the information needed to map memory into a multidimensional array. It is used internally by other parts of the framework to provide a uniform interface to internal data. The public Array class adds the domain information needed to support simple communication of subdomains. The communication routine source for operations such as halo and regrid are separated out into a separate directory to allow Arrays to be used internally before all the communication code has been compiled.
2. **LocalArray design.** The purpose of the LocalArray class is to fully describe a homogeneous multidimensional array, possibly strided, so that it can be understood and manipulated by multiple languages. It describes the relationship between array indices and the linear form of the array in physical memory. It describes all dimensions which are present; there are no hidden or implied dimensions. The first dimension specified is always the one which varies fastest in linearized memory regardless of interface language used to create or access the array.

   The LocalArray type is defined separately because it is used by the Fortran code in Fields, Grids, Route, and Regrid to refer to data independent of Type/Kind/Rank differences. This abstraction removes the need for these other objects to provide heavily-overloaded interface blocks to hide the number of different data combinations supported by these routines.

The metadata in this class would be unnecessary for a straight Fortran implementation since the language provides methods for querying arrays for this information. But for interoperability between different versions of Fortran, different hardware architectures, and the C++ interfaces it is necessary to keep the information in a format which can be easily managed by the ESMF and not buried in the language layer.

3. **Array design.** The purpose of the Array class is to support all the functions of the LocalArray class plus domain information to support halo, regrid, and data redistribution operations.

   The create routine in C++ requires the user to supply all values for rank, shape, strides, etc because there are no language constructs which allow a pointer to be queried for this information.

   There are two types of create routines in Fortran. One mimics the C++ interface and requires the user to supply all information. The second, which is expected to be more useful and natural, is simply passed an existing Fortran array pointer. Most of the array attributes can be queried using language-defined functions which should be portable to any Fortran compiler. If other attributes are needed which require compiler-dependent code, the implementation approach will be to write specific platform-dependent code for the most common compilers and platforms, and then use less efficient or more indirect methods for obtaining this information which will be the default if no compiler-specific method has been written.

   The major challenge for the Array class implementation is that it contains user data which can be of many different types, kinds, and ranks, each of which is a different type in Fortran 90 and the language is strictly typechecked.

   For the C++ interface polymorphism and templates can ease the burden of maintaining the interface; in Fortran the interface to the user is simplified by using interface blocks but the number of internal routines will be quite large. Judicious use of the macro preprocessor allows generic routines to be expanded on a per-datatype basis.

   Another way the data interfaces will be kept down to a manageable size is to explicitly limit the number of supported user datatypes to:

   integer*1/byte

   integer*2/short

   integer*4/int

   integer*8/long

   real*4/float

   real*8/double

4. **ArrayComm design.** The source of the ESMF_Array communication routines are separated out into an ESMF_ArrayComm directory. This allows basic Array functions to be compiled and used by Grid, Field, and Bundle code before the communication code has been compiled.

   The Array communication routines require Grid and DataMap information in addition to the Array itself. The ArrayDataMap is needed to identify which axes in the Array correspond to Grid axes. The Grid is needed to compute where in the overall object this Array is located and which pieces of the overall object are located on which DE.

22.5 Class API: Basic Array Methods

22.5.1 ESMF_ArrayGet

INTERFACE:

```
      subroutine ESMF_ArrayGet(array, rank, type, kind, counts, &
                               lbounds, ubounds, strides, haloWidth, &
                               base, name, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
integer, intent(out), optional :: rank
type(ESMF_DataType), intent(out), optional :: type
type(ESMF_DataKind), intent(out), optional :: kind
integer, dimension(:), intent(out), optional :: counts
integer, dimension(:), intent(out), optional :: lbounds
integer, dimension(:), intent(out), optional :: ubounds
integer, dimension(:), intent(out), optional :: strides
integer, intent(out), optional :: haloWidth
type(ESMF_Pointer), intent(out), optional :: base
character(len=ESMF_MAXSTR), intent(out), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return information about an `ESMF_Array`. For queries where the caller only wants a single value, specify the argument by name. All the arguments after the array input are optional to facilitate this.
The arguments are:

array An `ESMF_Array`.

[rank] The number of dimensions in the `array`.

[type] `ESMF_DataType`. Will be one of: `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, `ESMF_DATA_CHARACTER`, or `ESMF_DATA_COMPLEX`.

[kind] `ESMF_DataKind` variable which indicates the item size in bytes. Will be one of: `ESMF_I1`, `ESMF_I2`, `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, `ESMF_R8`, `ESMF_C8`, or `ESMF_C16`.

[counts] The number of items in each dimension of the `array`.

[lbounds] The lower index value of each dimension of the `array`.

[ubounds] The upper index value of each dimension of the `array`.

[strides] If nonzero, the spacing between index values per dimension of the `array`.

[haloWidth] Width of halo region.

[base] Base memory address of the data region of the `array`.

[name] `array` name. If one was not specified at create time, a unique name will have been generated.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.2 ESMF_ArrayGetAttribute - Retrieve a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetInt4Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an integer attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.3 ESMF_ArrayGetAttribute - Retrieve a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetInt4ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer list attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.4 ESMF_ArrayGetAttribute - Retrieve a 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetInt8Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an integer attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.5 ESMF_ArrayGetAttribute - Retrieve a 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetInt8ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte integer list attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.6 ESMF_ArrayGetAttribute - Retrieve a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetReal4Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.7 ESMF_ArrayGetAttribute - Retrieve a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetReal4ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from an `ESMF_Array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.8 ESMF_ArrayGetAttribute - Retrieve a 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetReal8Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte real attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.9 ESMF_ArrayGetAttribute - Retrieve a 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetReal8ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 8-byte real attribute from an `ESMF_Array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.10 ESMF_ArrayGetAttribute - Retrieve a logical attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetLogicalAttr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The logical value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.11 ESMF_ArrayGetAttribute - Retrieve a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetLogicalListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical list attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The logical values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.12 ESMF_ArrayGetAttribute - Retrieve a character attribute

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttribute()
subroutine ESMF_ArrayGetCharAttr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a character attribute from the `array`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to retrieve.

value The character value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.13 ESMF_ArrayGetAttributeCount - Query the number of attributes

INTERFACE:

```
subroutine ESMF_ArrayGetAttributeCount(array, count, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
integer, intent(out) :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the number of attributes associated with the given `array` in the argument `count`.
The arguments are:

array An `ESMF_Array` object.

count The number of attributes associated with this object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.14 ESMF_ArrayGetAttributeInfo - Query Field attributes by name

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttributeInfo()
subroutine ESMF_ArrayGetAttrInfoByName(array, name, datatype, &
                                        datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character(len=*), intent(in) :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the named attribute, including `datatype` and `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to query.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] The number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.15 ESMF_ArrayGetAttributeInfo - Query Field attributes by index number

INTERFACE:

```
! Private name; call using ESMF_ArrayGetAttributeInfo()
subroutine ESMF_ArrayGetAttrInfoByNum(array, attributeIndex, name, &
                                       datatype, datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
integer, intent(in) :: attributeIndex
character(len=*), intent(out), optional :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed attribute, including `name`, `datatype`, `datakind` (if applicable) and `count`.
The arguments are:

array An `ESMF_Array` object.

attributeIndex The index number of the attribute to query.

name Returns the name of the attribute.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

[count] Returns the number of items in this attribute. For character types, this is the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.16 ESMF_ArrayPrint - Print contents of an Array object

INTERFACE:

```
subroutine ESMF_ArrayPrint(array, options, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `array` to `stdout`.
The arguments are:

array An `ESMF_Array`.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.17 ESMF_ArraySetAttribute - Set a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetInt4Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The integer value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.18 ESMF_ArraySetAttribute - Set a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetInt4ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer list attribute to the `array`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.19 ESMF_ArraySetAttribute - Set a 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetInt8Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The integer value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.20 ESMF_ArraySetAttribute - Set a 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetInt8ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer list attribute to the `array`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.21 ESMF_ArraySetAttribute - Set a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetReal4Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.22 ESMF_ArraySetAttribute - Set a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetReal4ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real list attribute to the `array`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.23 ESMF_ArraySetAttribute - Set a 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetReal8Attr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte real attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.24 ESMF_ArraySetAttribute - Set a 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetReal8ListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte real list attribute to the `array`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.25 ESMF_ArraySetAttribute - Set a logical attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetLogicalAttr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The logical true/false value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.26 ESMF_ArraySetAttribute - Set a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetLogicalListAttr(array, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical list attribute to the `array`. The attribute has a `name` and a `valueList`. The number of logical items in the `valueList` is given by `count`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

count The number of logicals in the `valueList`.

value The logical true/false values of the attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.27 ESMF_ArraySetAttribute - Set a character attribute

INTERFACE:

```
! Private name; call using ESMF_ArraySetAttribute()
subroutine ESMF_ArraySetCharAttr(array, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
character (len = *), intent(in) :: name
character (len = *), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a character attribute to the `array`. The attribute has a `name` and a `value`.
The arguments are:

array An `ESMF_Array` object.

name The name of the attribute to add.

value The character value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.28 ESMF_ArraySet - Set information about an Array

INTERFACE:

```
subroutine ESMF_ArraySet(array, name, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
character (len = *), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the name of the `ESMF_Array`. Note: Unlike most other ESMF objects there are very few items which can be changed once an `ESMF_Array` is created.
The arguments are:

array An `ESMF_Array`.

[name] The array name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.29 ESMF_ArrayValidate - Check validity of an Array

INTERFACE:

```
subroutine ESMF_ArrayValidate(array, options, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `array` is internally consistent. Currently this method determines if the `array` has a valid data pointer. The method returns an error code if problems are found.
The arguments are:

array An `ESMF_Array`.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.5.30 ESMF_ArrayWrite

INTERFACE:

```
subroutine ESMF_ArrayWrite(array, iospec, filename, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
type(ESMF_IOSpec), intent(in), optional :: iospec
character(len=*), intent(in), optional :: filename
integer, intent(out), optional :: rc
```

DESCRIPTION:

Used to write data to persistent storage in a variety of formats. (see writerestart/restore for quick data dumps.) Details of I/O options specified with an `ESMF_IOSpec`.
The arguments are:

array An `ESMF_Array`.

[iospec] The file specification.

[filename] The file name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.6 Class API: Array Overloads for Fortran Arrays

22.6.1 ESMF_ArrayCreate – Generic interface to create an Array

INTERFACE:

```
interface ESMF_ArrayCreate
```

PRIVATE MEMBER FUNCTIONS:

```
module procedure ESMF_ArrayCreateByList ! specify TKR
module procedure ESMF_ArrayCreateBySpec ! specify ArraySpec
! Plus interfaces for each T/K/R
```

This interface provides a single (heavily overloaded) entry point for the various types of ESMF_ArrayCreate functions.
There are 3 options for setting the contents of the ESMF_Array at creation time:

Allocate Space Only Data space is allocated but not initialized. The caller can query for a pointer to the start of the space to address it directly. The caller must not deallocate the space; the ESMF_Array will release the space when it is destroyed.

Data Copy An existing Fortran array is specified and the data contents are copied into new space allocated by the ESMF_Array. The caller must not deallocate the space; the ESMF_Array will release the space when it is destroyed.

Data Reference An existing Fortran array is specified and the data contents reference it directly. The caller is responsible for deallocating the space; when the ESMF_Array is destroyed it will not release the space.

There are 3 options for specifying the type/kind/rank of the ESMF_Array data:

List The characteristics of the ESMF_Array are given explicitly by individual arguments to the create function.

ArraySpec A previously created ESMF_ArraySpec object is given which describes the characteristics.

Fortran 90 Pointer An associated or unassociated Fortran 90 array pointer is used to describe the array. (Only available from the Fortran interface.)

The concept of an "empty" Array does not exist. To make an ESMF object which stores the Type/Kind/Rank information create an ESMF_ArraySpec object which can then be used repeatedly in subsequent Array Create calls.
end interface

22.6.2 ESMF_ArrayCreate - Make an ESMF array from an allocated Fortran array

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateByFullPtr<rank><type><kind>(farr, docopy, haloWidth, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateByFullPtr<rank><type><kind>
```

ARGUMENTS:

```
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farr
type(ESMF_CopyFlag), intent(in), optional :: docopy
integer, intent(in), optional :: haloWidth
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Array` based on an already allocated Fortran array pointer. This routine can make a copy or reference the existing data and saves all necessary information about bounds, data type, kind, etc. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The function return is an `ESMF_Array` type.
The arguments are:

farr An allocated Fortran array pointer.

[docopy] Default to `ESMF_DATA_REF`, makes the `ESMF_Array` reference the existing data array. If set to `ESMF_DATA_COPY` this routine allocates new space and copies the data from the pointer into the new array.

[haloWidth] Set the maximum width of the halo region on all edges. Defaults to 0.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.6.3 ESMF_ArrayCreate – Create an Array specifying all options.

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateByList(rank, type, kind, counts, &
                                haloWidth, lbounds, ubounds, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateByList
```

ARGUMENTS:

```
integer, intent(in) :: rank
type(ESMF_DataType), intent(in) :: type
type(ESMF_DataKind), intent(in) :: kind
integer, dimension(:), intent(in) :: counts
integer, intent(in), optional :: haloWidth
integer, dimension(:), intent(in), optional :: lbounds
integer, dimension(:), intent(in), optional :: ubounds
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a new `ESMF_Array` and allocate data space, which remains uninitialized. The return value is the new `ESMF_Array`.
The arguments are:

rank Array rank (dimensionality – 1D, 2D, etc). Maximum allowed is 7D.

type Array type. Valid types include ESMF_DATA_INTEGER, ESMF_DATA_REAL, ESMF_DATA_LOGICAL, or ESMF_DATA_CHARACTER.

kind Array kind. Valid kinds include ESMF_KIND_I4, ESMF_KIND_I8, ESMF_KIND_R4, ESMF_KIND_R8, ESMF_KIND_C8, ESMF_KIND_C16.

counts The number of items in each dimension of the array. This is a 1D integer array the same length as the rank.

[haloWidth] Set the maximum width of the halo region on all edges. Defaults to 0.

[lbounds] An integer array of length rank with the lower index for each dimension.

[ubounds] An integer array of length rank with the upper index for each dimension.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.6.4 ESMF_ArrayCreate - Make an ESMF array from an unallocated Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateByMTPtr<rank><type><kind>(farr, counts, haloWidth, lbounds, ubo
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateByMTPtr<rank><type><kind>
```

ARGUMENTS:

```
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farr
integer, dimension(:), intent(in) :: counts
integer, intent(in), optional :: haloWidth
integer, dimension(:), intent(in), optional :: lbounds
integer, dimension(:), intent(in), optional :: ubounds
integer, intent(out), optional :: rc
```

DESCRIPTION:

Creates an ESMF_Array based on an unallocated (but allocatable) Fortran array pointer. This routine allocates memory to the array and saves all necessary information about bounds, data type, kind, etc. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The function return is an ESMF_Array type with space allocated for data.
The arguments are:

farr An allocatable (but currently unallocated) Fortran array pointer.

counts An integer array of counts. Must be the same length as the rank.

[haloWidth] An integer count of the width of the halo region on all sides of the array. The default is 0, no halo region.

[lbounds] An integer array of lower index values. Must be the same length as the rank.

[ubounds] An integer array of upper index values. Must be the same length as the rank.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.6.5 ESMF_ArrayCreate – Create a new Array from an ArraySpec

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateBySpec(arrayspec, counts, haloWidth, &
                                lbounds, ubounds, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateBySpec
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec
integer, intent(in), dimension(:) :: counts
integer, intent(in), optional :: haloWidth
integer, dimension(:), intent(in), optional :: lbounds
integer, dimension(:), intent(in), optional :: ubounds
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a new ESMF_Array and allocate data space, which remains uninitialized. The return value is the new ESMF_Array.
The arguments are:

arrayspec An ESMF_ArraySpec object which contains the type/kind/rank information for the data.

counts The number of items in each dimension of the array. This is a 1D integer array the same length as the rank.

[haloWidth] Set the maximum width of the halo region on all edges. Defaults to 0.

[lbounds] An integer array of length rank with the lower index for each dimension.

[ubounds] An integer array of length rank, with the upper index for each dimension.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

INTERFACE:

```
subroutine ESMF_ArrayDestroy(array, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this ESMF_Array.
The arguments are:

array Destroy contents of this ESMF_Array.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

To reduce the depth of crossings of the F90/C++ boundary we first query to see if we are responsible for deleting the data space. If so, first deallocate the space and then call the C++ code to release the object space. When it returns we are done and can return to the user. Otherwise we would need to make a nested call back into F90 from C++ to do the deallocation during the object delete.

22.6.6 ESMF_ArrayGetData - Retrieve a Fortran pointer to Array data

INTERFACE:

```
! Private name; call using ESMF_ArrayGetData()
subroutine ESMF_ArrayGetData<rank><type><kind>(array, fptr, docopy, rc)
```

ARGUMENTS:

```
type(ESMF_Array) :: array
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: fptr
type(ESMF_CopyFlag), intent(in), optional :: docopy
integer, intent(out), optional :: rc
```

DESCRIPTION:

Given an `ESMF_Array` return a Fortran pointer to the existing data buffer, or return a Fortran pointer to a new copy of the data. Valid type/kind/rank combinations supported by the framework are: ranks 1 to 7, type real of kind *4 or *8, and type integer of kind *1, *2, *4, or *8.
The arguments are:

array An `ESMF_Array`.

farr An allocatable (but currently unallocated) Fortran array pointer.

docopy Default to `ESMF_DATA_REF`, makes the `ESMF_Array` reference the existing data array. If set to `ESMF_DATA_COPY` this routine allocates new space and copies the data from the pointer into the space.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

22.7 Class API: Array Communications

22.7.1 ESMF_ArrayGather - Gather an Array onto one DE

INTERFACE:

```
subroutine ESMF_ArrayGather(array, grid, datamap, rootDE, gatheredArray, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
type(ESMF_Grid), intent(in) :: grid
type(ESMF_FieldDataMap), intent(in) :: datamap
integer, intent(in) :: rootDE
type(ESMF_Array), intent(out) :: gatheredArray
integer, intent(out), optional :: rc
```

DESCRIPTION:

Gather a distributed `ESMF_Array` over multiple DEs into a single `ESMF_Array` on one DE.
The arguments are:

array `ESMF_Array` containing distributed data to be gathered.

grid `ESMF_Grid` which corresponds to the distributed data.

datamap `ESMF_FieldDataMap` which describes the mapping of the data onto the cells in the `ESMF_Grid`.

rootDE The DE number on which the resulting gathered ESMF_Array will be created.

gatheredArray On the rootDE, the resulting gathered ESMF_Array. On all other DEs, an invalid ESMF_Array.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.2 ESMF_ArrayHalo - Halo an Array

INTERFACE:

```
! Private name; call using ESMF_ArrayHalo()
subroutine ESMF_ArrayHaloNew(array, routehandle, blocking, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
type(ESMF_RouteHandle), intent(in) :: routehandle
type(ESMF_BlockingFlag), intent(in), optional :: blocking
type(ESMF_CommHandle), intent(inout), optional :: commhandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Perform a halo operation over the data in an ESMF_Array. This routine updates the data inside the ESMF_Array in place. It uses a precomputed ESMF_Route for the communications pattern. (See ESMF_ArrayHaloPrecompute() for how to precompute and associate an ESMF_Route with an ESMF_RouteHandle).

array ESMF_Array containing data to be haloed.

routehandle ESMF_RouteHandle which was returned from an ESMF_ArrayHaloPrecompute() call.

[blocking] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communications. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blocking flag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.3 ESMF_ArrayHaloRelease - Release resources stored for halo operation

INTERFACE:

```
subroutine ESMF_ArrayHaloRelease(routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

When the precomputed information about a halo operation is no longer needed, this routine releases the associated resources.

routehandle ESMF_RouteHandle associated with halo operation which should be released.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.4 ESMF_ArrayHaloStore - Store resources for a halo operation

INTERFACE:

```
subroutine ESMF_ArrayHaloStore(array, grid, datamap, routehandle, &
                               halodirection, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
type(ESMF_Grid), intent(in) :: grid
type(ESMF_FieldDataMap), intent(in) :: datamap
type(ESMF_RouteHandle), intent(out) :: routehandle
type(ESMF_HaloDirection), intent(in), optional :: halodirection
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute the data movements needed to perform a halo operation over the data in an ESMF_Array. It associates this information with the routehandle, which should then provided to ESMF_ArrayHalo() at execution time. The ESMF_Grid and ESMF_FieldDataMap are used as templates to understand how this ESMF_Array relates to ESMF_Arrays on other DEs.

array ESMF_Array containing data to be haloed.

grid ESMF_Grid which matches how this data was decomposed.

datamap ESMF_FieldDataMap which matches how the data in the ESMF_Array relates to the given ESMF_Grid.

routehandle ESMF_RouteHandle is returned to be used during the execution of the halo operation.

[halodirection] ESMF_HaloDirection to indicate which of the boundaries should be updated. If not specified, all boundaries are updated.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.5 ESMF_ArrayRedist - Redistribute an Array

INTERFACE:

```
! Private name; call using ESMF_ArrayRedist
subroutine ESMF_ArrayRedistNew(srcArray, dstArray, routehandle, &
                               blocking, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: srcArray
type(ESMF_Array), intent(inout) :: dstArray
type(ESMF_RouteHandle), intent(in) :: routehandle
type(ESMF_BlockingFlag), intent(in), optional :: blocking
type(ESMF_CommHandle), intent(inout), optional :: commhandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Redistribute the data in one set of ESMF_Arrays to another set of ESMF_Arrays. Data redistribution does no interpolation, so during the ESMF_ArrayRedistPrecompute() call the ESMF_Grids must have identical coordinates. The distribution of the ESMF_Grid can be over different ESMF_DELayouts, or the ESMF_FieldDataMaps can differ. The routehandle argument must be the one which was associated with the precomputed data movements during the precompute operation, and if the data movement is identical for different collections of ESMF_Arrays, the same routehandle can be supplied during multiple calls to this execution routine, specifying a different set of source and destination ESMF_Arrays each time.

srcArray ESMF_Array containing source data.

dstArray ESMF_Array containing results.

routehandle ESMF_RouteHandle precomputed by ESMF_ArrayRedistPrecompute().

[blocking] Optional argument which specifies whether the operation should wait until complete before returning or return as soon as the communication between DEs has been scheduled. If not present, default is to do synchronous communications. Valid values for this flag are ESMF_BLOCKING and ESMF_NONBLOCKING. (This feature is not yet supported. All operations are synchronous.)

[commhandle] If the blocking flag is set to ESMF_NONBLOCKING this argument is required. Information about the pending operation will be stored in the ESMF_CommHandle and can be queried or waited for later.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.6 ESMF_ArrayRedistRelease - Release resources stored for redist operation

INTERFACE:

```
subroutine ESMF_ArrayRedistRelease(routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_RouteHandle), intent(inout) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

When the precomputed information about a redistribution operation is no longer needed, this routine releases the associated resources.

routehandle ESMF_RouteHandle associated with redist operation which should be released.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

22.7.7 ESMF_ArrayRedistStore - Store resources for a redist operation

INTERFACE:

```
subroutine ESMF_ArrayRedistStore(srcArray, srcGrid, srcDataMap, &
                                 dstArray, dstGrid, dstDataMap, &
                                 parentDElayout, routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: srcArray
type(ESMF_Grid), intent(in) :: srcGrid
type(ESMF_FieldDataMap), intent(in) :: srcDataMap
type(ESMF_Array), intent(inout) :: dstArray
type(ESMF_Grid), intent(in) :: dstGrid
type(ESMF_FieldDataMap), intent(in) :: dstDataMap
type(ESMF_DELayout), intent(in) :: parentDElayout
type(ESMF_RouteHandle), intent(out) :: routehandle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Precompute and associate the required data movements to redistribute data over one set of `ESMF_Array`s to another set of `ESMF_Array`s. Data redistribution does no interpolation, so both `ESMF_Grid`s must have identical coordinates. The distribution of the `ESMF_Grid`s can be over different `ESMF_DELayout`s, or the `ESMF_FieldDataMap`s can differ. The `routehandle` argument is associated with the stored information and must be supplied to `ESMF_ArrayRedist()` to execute the operation. Call `ESMF_ArrayRedistRelease()` when this information is no longer required.
The arguments are:

srcArray `ESMF_Array` containing the data source.

srcGrid `ESMF_Grid` describing the grid on which the source data is arranged.

srcDataMap `ESMF_FieldDataMap` describing how the source data maps onto the grid.

dstArray `ESMF_Array` where the destination data will be put.

dstGrid `ESMF_Grid` describing the grid on which the destination data is arranged.

dstDataMap `ESMF_FieldDataMap` describing how the destination data maps onto the grid.

parentDElayout `ESMF_DELayout` object which includes all DEs in both the source and destination grids.

routehandle Returned `ESMF_RouteHandle` which identifies this communication pattern.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

23 ArrayDataMap Class

23.1 Description

The ArrayDataMap class maintains information about the way an ESMF Array object maps to the associated Grid in a Field object. The Array class maintains the linearization of multidimensional data to memory addresses, but the ArrayDataMap class maintains the mapping between rank numbers and Grid axes, and contains counts for dimensions which do not correspond to a Grid axis.
The ArrayDataMap class implements methods for reordering or repacking memory which can be transparent to the calling code. None of the methods in the ArrayDataMap class change data values; they simply change the mapping to memory addresses.

23.2 ArrayDataMap Options

23.2.1 ESMF_IndexOrder

DESCRIPTION:
A set of predefined index orders which shortcut setting the mapping between data and grid indices.

ESMF_INDEX_I One dimensional data and grid.

ESMF_INDEX_IJ Two dimensional, IJ ordering.

ESMF_INDEX_JI Two dimensional, JI ordering.

ESMF_INDEX_IJK Three dimensional, IJK ordering.

ESMF_INDEX_JIK Three dimensional, JIK ordering.

ESMF_INDEX_KJI Three dimensional, KJI ordering.

ESMF_INDEX_IKJ Three dimensional, IKJ ordering.

ESMF_INDEX_JKI Three dimensional, JKI ordering.

ESMF_INDEX_KIJ Three dimensional, KIJ ordering.

23.2.2 ESMF_RelLoc

DESCRIPTION:
Description of how data items are located relative to an individual cell or element in the grid. (See the `ESMF_Grid` documentation for a description of 'staggering' which is a per-grid concept.)
Valid values are:

ESMF_CELL_UNDEFINED Data location is undefined.

ESMF_CELL_CENTER Data location is at cell center.

ESMF_CELL_NFACE Data location is on north face of cell.

ESMF_CELL_SFACE Data location is on south face of cell.

ESMF_CELL_EFACE Data location is on east face of cell.

ESMF_CELL_WFACE Data location is on west face of cell.

ESMF_CELL_NECORNER Data location is at north-east corner of cell.

ESMF_CELL_NWCORNER Data location is at north-west corner of cell.

ESMF_CELL_SECORNER Data location is at south-east corner of cell.

ESMF_CELL_SWCORNER Data location is at south-west corner of cell.

ESMF_CELL_TOPFACE Data location is on top face of cell.

ESMF_CELL_BOTFACE Data location is on bottom face of cell.

ESMF_CELL_CELL Data location is over entire cell.

ESMF_CELL_VERTEX Data location is at the vertices of the cell.

23.3 Use and Examples

ArrayDataMaps are shallow objects. They can be declared as local (stack) variables in subroutines. They do not need a create or destroy method. There is a method to set initial values, to set and query individual values, and to print the contents in human-readable form for output or debugging.

```
! !PROGRAM: ESMF_ArrayDataMapEx - Array DataMap manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Array DataMap set and get usage
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! local variables
      type(ESMF_ArrayDataMap) :: arrayDM
      integer :: drank
      integer :: dlist(ESMF_MAXDIM), dcounts(ESMF_MAXDIM)


      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(rc=rc)
```

23.3.1 Setting Array DataMap Defaults and Invalidation

This example shows how to set the default values in an `ESMF_ArrayDataMap`, and how to intentionally mark an `ESMF_ArrayDataMap` invalid.

```
      ! Set up a default data map for a Array with 2D data,
      ! and a 1-for-1 mapping with the Grid.
      call ESMF_ArrayDataMapSetDefault(arrayDM, 2, rc=rc)


      print *, "Default values for ArrayDataMap = "
      call ESMF_ArrayDataMapPrint(arrayDM, rc=rc)


      call ESMF_ArrayDataMapSetInvalid(arrayDM, rc=rc)


      print *, "Invalid ArrayDataMap = "
      call ESMF_ArrayDataMapPrint(arrayDM, rc=rc)
```

23.3.2 Setting Array DataMap Values

This example shows how to set values in an `ESMF_ArrayDataMap`.

```
dlist(1:3) = (/ 1, 2, 0 /)
dcounts(1) = 4
call ESMF_ArrayDataMapSet(arrayDM, dataRank=3, dataIndexList=dlist, &
                          counts=dcounts, rc=rc)


print *, "ArrayDataMap after set = "
call ESMF_ArrayDataMapPrint(arrayDM, rc=rc)
```

23.3.3 Getting Array DataMap Values

This example shows how to query an `ESMF_ArrayDataMap`.

```
call ESMF_ArrayDataMapGet(arrayDM, drank, dlist, dcounts, rc=rc)
print *, "Returned values from Array DataMap:"
print *, "rank =", drank
print *, "correspondance to grid indices = ", dlist
print *, "counts for non-grid dimensions =", dcounts


! finalize ESMF framework
call ESMF_Finalize(rc)


end program ESMF_ArrayDataMapEx
```

23.4 Restrictions and Future Work

1. **Native C++ vs. Fortran index ordering is not supported yet.** ESMF currently assumes that all arrays are allocated from Fortran.
2. **Complex interleave is not supported yet.** ESMF does not yet support a complex type. When we do, we expect that the user will be able to specify either an all-real followed by all-imaginary format or a real/imaginary interleaved format.

23.5 Class API

23.5.1 ESMF_ArrayDataMapGet - Get values from an ArrayDataMap

INTERFACE:

```
subroutine ESMF_ArrayDataMapGet(arraydatamap, dataRank, dataIndexList, &
                                counts, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap), intent(in) :: arraydatamap
integer, intent(out), optional :: dataRank
integer, dimension(:), intent(out), optional :: dataIndexList
integer, dimension(:), intent(out), optional :: counts
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return information from an ESMF_ArrayDataMap.
The arguments are:

arraydatamap An ESMF_ArrayDataMap.

[datarank] The number of dimensions in the data ESMF_Array.

[dataIndexList] An integer array, datarank long, which specifies the mapping between rank numbers in the ESMF_Grid and the ESMF_Array. If there is no correspondance (because the ESMF_Array has a higher rank than the ESMF_Grid) the index value will be 0.

[counts] An integer array, with length (datarank minus the grid rank). Each entry is the default item count which would be used for those ranks which do not correspond to grid ranks when creating an ESMF_Field using only an an ESMF_ArraySpec and an ESMF_ArrayDataMap.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.5.2 ESMF_ArrayDataMapPrint - Print an ArrayDataMap

INTERFACE:

```
subroutine ESMF_ArrayDataMapPrint(arraydatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap), intent(in) :: arraydatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the arraydatamap to stdout.
The arguments are:

arraydatamap ESMF_ArrayDataMap to print.

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.5.3 ESMF_ArrayDataMapSet - Set values in an ArrayDataMap

INTERFACE:

```
subroutine ESMF_ArrayDataMapSet(arraydatamap, dataRank, &
                                dataIndexList, counts, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap), intent(inout) :: arraydatamap
integer, intent(in), optional :: dataRank
integer, dimension(:), intent(in), optional :: dataIndexList
integer, dimension(:), intent(in), optional :: counts
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set values in an ESMF_ArrayDataMap.
The arguments are:

arraydatamap An ESMF_ArrayDataMap.

[datarank] The number of dimensions in the data ESMF_Array.

[dataIndexList] An integer array, datarank long, which specifies the mapping between rank numbers in the ESMF_Grid and the ESMF_Array. If there is no correspondance (because the ESMF_Array has a higher rank than the ESMF_Grid) the index value must be 0.

For example, if the data array is 3D and the grid is 2D, and the first and second data indices correspond to the grid and the last data index is to store duplicate values for the same grid location, then the value for this input would be 1, 2, 0. If the middle index was the one which was for the duplicate values, it would be 1, 0, 2.

[counts] An integer array, with length (datarank minus the grid rank). If the ESMF_Array is a higher rank than the ESMF_Grid, the additional dimensions may optionally each have an item count defined here. This allows ESMF_FieldCreate() to take an ESMF_ArraySpec and an ESMF_ArrayDataMap and create the appropriately sized ESMF_Array for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if ESMF_FieldCreate() is called called with an already-created ESMF_Array.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.5.4 ESMF_ArrayDataMapSetDefault - Set ArrayDataMap default values

INTERFACE:

```
! Private name; call using ESMF_ArrayDataMapSetDefault()
subroutine ESMF_ArrayDataMapSetDefExplicit(arraydatamap, dataRank, &
                                           dataIndexList, counts, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap) :: arraydatamap
integer, intent(in) :: dataRank
integer, dimension(:), intent(in), optional :: dataIndexList
integer, dimension(:), intent(in), optional :: counts
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set default values of an ESMF_ArrayDataMap. This differs from ESMF_ArrayDataMapSet() in that all values which are not specified here will be overwritten with default values.

arraydatamap An ESMF_ArrayDataMap.

datarank The number of dimensions in the data ESMF_Array.

[dataIndexList] An integer array, datarank long, which specifies the mapping between rank numbers in the ESMF_Grid and the ESMF_Array. If there is no correspondance (because the ESMF_Array has a higher rank than the ESMF_Grid) the index value must be 0. The default is a 1-to-1 mapping with the ESMF_Grid.

For example, if the data array is 3D and the grid is 2D, and the first and second data indices correspond to the grid and the last data index is to store duplicate values for the same grid location, then the value for this input would be 1, 2, 0. If the middle index was the one which was for the duplicate values, it would be 1, 0, 2. The default values for this are 1, 0, ... for a 1D grid and 1, 2, 0, ... for a 2D grid.

[counts] An integer array, with length (datarank minus the grid rank). If the ESMF_Array is a higher rank than the ESMF_Grid, the additional dimensions may optionally each have an item count defined here. This allows ESMF_FieldCreate() to take an ESMF_ArraySpec and an ESMF_ArrayDataMap and create the appropriately sized ESMF_Array for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if ESMF_FieldCreate() is called called with an already-created ESMF_Array. If unspecified, the default lengths are 1.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.5.5 ESMF_ArrayDataMapSetDefault - Set ArrayDataMap default values

INTERFACE:

```
! Private name; call using ESMF_ArrayDataMapSetDefault()
subroutine ESMF_ArrayDataMapSetDefIndex(arraydatamap, indexorder, counts, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap) :: arraydatamap
type(ESMF_IndexOrder), intent(in) :: indexorder
integer, dimension(:), intent(in), optional :: counts
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set default values of an ESMF_ArrayDataMap. This differs from ESMF_ArrayDataMapSet() in that all values which are not specified here will be overwritten with default values.

arraydatamap An ESMF_ArrayDataMap.

indexorder An ESMF_DataIndexOrder which specifies one of several common predefined mappings between the grid and data ranks. This is simply a convenience for the common cases; there is a more general form of this call which allows the mapping to be specified as an integer array of index numbers directly.

[counts] An integer array, with length (datarank minus the grid rank). If the ESMF_Array is a higher rank than the ESMF_Grid, the additional dimensions may optionally each have an item count defined here. This allows ESMF_FieldCreate() to take an ESMF_ArraySpec and an ESMF_ArrayDataMap and create the appropriately sized ESMF_Array for each DE. These values are unneeded if the ranks of the data and grid are the same, and ignored if ESMF_FieldCreate() is called called with an already-created ESMF_Array. If unspecified, the default lengths are 1.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.5.6 ESMF_ArrayDataMapSetInvalid - Set ArrayDataMap to invalid status

INTERFACE:

```
subroutine ESMF_ArrayDataMapSetInvalid(arraydatamap, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap), intent(inout) :: arraydatamap
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set the contents of an `ESMF_ArrayDataMap` to an uninitialized value.
The arguments are:

arraydatamap An `ESMF_ArrayDataMap`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

23.5.7 ESMF_ArrayDataMapValidate - Check validity of an ArrayDataMap

INTERFACE:

```
subroutine ESMF_ArrayDataMapValidate(arraydatamap, options, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayDataMap), intent(in) :: arraydatamap
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `arraydatamap` is internally consistent. Currently this method determines if the `arraydatamap` is set up for use. The method returns an error code if problems are found.
The arguments are:

arraydatamap `ESMF_ArrayDataMap` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

24 ArraySpec Class

24.1 Description

An ArraySpec is a very simple class that contains type, kind, and rank information about an array. For those unfamiliar with Fortran:

- **Type** describes the data type of the elements in the array, such as integer, real, logical, etc.;
- **Kind** describes their precision; and
- **Rank** refers to their dimensionality.

The only methods that are associated with the ArraySpec class are those that allow you to set and retrieve this type, kind, and rank information.

24.2 Use and Examples

The ArraySpec is passed in as an argument at Field and Bundle creation in order to describe an Array that will be allocated or attached at a later time. There are any number of situations in which this approach is useful. One common example is a case in which the user wants to create a very flexible export State with many diagnostic variables predefined, but only a subset desired and consequently allocated for a particular run.

```
! !PROGRAM: ESMF_ArraySpecEx - ArraySpec manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of ArraySpec set and get usage
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! local variables
      type(ESMF_ArraySpec) :: arrayDS
      integer :: myrank
      type(ESMF_DataType) :: mytype
      type(ESMF_DataKind) :: mykind


      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(rc=rc)
```

24.2.1 Setting ArraySpec Values

This example shows how to set values in an ESMF_ArraySpec.

```
      call ESMF_ArraySpecSet(arrayDS, rank=2, type=ESMF_DATA_REAL, &
                             kind=ESMF_R8, rc=rc)
```

24.2.2 Getting ArraySpec Values

This example shows how to query an ESMF_ArraySpec.

```
      call ESMF_ArraySpecGet(arrayDS, myrank, mytype, mykind, rc)
      print *, "Returned values from ArraySpec:"
      print *, "rank =", myrank


      ! finalize ESMF framework
      call ESMF_Finalize(rc)


      end program ESMF_ArraySpecEx
```

24.3 Restrictions and Future Work

1. **Limit on rank.** The values for type, kind and rank passed into the ArraySpec class are subject to the same limitations as Arrays. The maximum array rank is 7, which is the highest rank supported by Fortran.

24.4 Design and Implementation Notes

The information contained in an ESMF_ArraySpec is used to create ESMF_Array objects.
ESMF_ArraySpec is a shallow class, and only set and get methods are needed. They do not need to be created or destroyed.

24.5 Class API

24.5.1 ESMF_ArraySpecGet - Get values from an ArraySpec

INTERFACE:

```
subroutine ESMF_ArraySpecGet(arrayspec, rank, type, kind, rc)
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec
integer, intent(out), optional :: rank
type(ESMF_DataType), intent(out), optional :: type
type(ESMF_DataKind), intent(out), optional :: kind
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information about the contents of an ESMF_ArraySpec.
The arguments are:

arrayspec The ESMF_ArraySpec to query.

rank ESMF_Array rank (dimensionality – 1D, 2D, etc). Maximum possible is 7D.

type ESMF_Array type. Valid types include ESMF_DATA_INTEGER, ESMF_DATA_REAL, ESMF_DATA_LOGICAL, ESMF_DATA_CHARACTER.

kind ESMF_Array kind. Valid kinds include ESMF_KIND_I4, ESMF_KIND_I8, ESMF_KIND_R4, ESMF_KIND_R8, ESMF_KIND_C8, ESMF_KIND_C16.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

24.5.2 ESMF_ArraySpecSet - Set values for an ArraySpec using type,kind,rank

INTERFACE:

```
! Private name; call using ESMF_ArraySpecSet()
subroutine ESMF_ArraySpecSetThree(arrayspec, rank, type, kind, rc)
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(inout) :: arrayspec
integer, intent(in) :: rank
type(ESMF_DataType), intent(in) :: type
type(ESMF_DataKind), intent(in) :: kind
integer, intent(out), optional :: rc
```

DESCRIPTION:

Creates a description of the data – the type, the dimensionality, etc. This specification can be used in an ESMF_ArrayCreate call with data to create a full ESMF_Array.
The arguments are:

arrayspec The ESMF_ArraySpec to set.

rank Array rank (dimensionality – 1D, 2D, etc). Maximum allowed is 7D.

type ESMF_Array type. Valid types include ESMF_DATA_INTEGER, ESMF_DATA_REAL, ESMF_DATA_LOGICAL, ESMF_DATA_CHARACTER.

kind ESMF_Array kind. Valid kinds include ESMF_KIND_I4, ESMF_KIND_I8, ESMF_KIND_R4, ESMF_KIND_R8, ESMF_KIND_C8, ESMF_KIND_C16.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

e

25 Grid Class

25.1 Description

The ESMF Grid class represents all aspects of the computational domain and its decomposition in a parallel-processing environment, and must provide access to any necessary grid information to the rest of the ESMF. The ESMF Grid class is included in the Field class and the Gridded Component class and provides information needed in data communication methods like halo and redistribution. It has methods to internally generate a variety of simple grids or read in more complicated grids provided by a user (reading in grids is not yet implemented). The ESMF Grid class supports multi-component coupling by providing a common structure necessary for regridding.
A single Grid can have more than one related subGrid. Each subGrid corresponds to a different representation of the Grid. For example, a staggered Grid could have separate subGrids representing locations at the cell centers and cell faces. A vertical grid may also be represented as a subGrid.

25.2 Grid Options

25.2.1 ESMF_CoordIndex

DESCRIPTION:
The Grid class can be set to different default indexing of coordinates. This parameter describes indexing options supported by ESMF.
Valid values are:

ESMF_COORD_INDEX_GLOBAL The coordinates are indexed globally.

ESMF_COORD_INDEX_LOCAL The coordinates are indexed locally.

ESMF_COORD_INDEX_UNKNOWN Unknown or undefined coordinate indexing.

25.2.2 ESMF_CoordOrder

DESCRIPTION:
The Grid class can be set to different default ordering of coordinates, for example KIJ. This parameter describes ordering options supported by ESMF.
Valid values are:

ESMF_COORD_ORDER_UNKNOWN Unknown or undefined coordinate ordering.

ESMF_COORD_ORDER_XYZ The coordinates are ordered XYZ. This is equivalent to IJK ordering. For a 2D grid, this defaults to IJ mapping.

ESMF_COORD_ORDER_XZY The coordinates are ordered XZY. For a 2D grid, this defaults to IJ mapping to XY.

ESMF_COORD_ORDER_YXZ IJK maps to YXZ. For a 2D grid, this defaults to IJ mapping to YX.

ESMF_COORD_ORDER_YZX IJK maps to YZX.

ESMF_COORD_ORDER_ZXY IJK maps to ZXY.

ESMF_COORD_ORDER_ZYX IJK maps to ZYX.

25.2.3 ESMF_CoordSystem

DESCRIPTION:
Supported coordinate systems.
Valid values are:

ESMF_COORD_SYSTEM_CARTESIAN Cartesian coordinates (x,y).

ESMF_COORD_SYSTEM_CYLINDRICAL Cylindrical coordinates.

ESMF_COORD_SYSTEM_DEPTH Vertical z coordinate depth (0 at top surface).

ESMF_COORD_SYSTEM_HEIGHT Vertical z coordinate height (0 at bottom).

ESMF_COORD_SYSTEM_SPHERICAL Spherical coordinates (longitude, latitude).

ESMF_COORD_SYSTEM_UNKNOWN Unknown or undefined coordinate system.

ESMF_COORD_SYSTEM_USER User-defined coordinate system.

25.2.4 ESMF_GridHorzStagger

DESCRIPTION:
Horizontal Grid staggerings supported by ESMF.
Valid values are:

ESMF_GRID_HORZ_STAGGER_A Arakawa A staggering where all fields, including velocities, are located at cell centers. A grid created with this staggering will only accept `ESMF_Fields` with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER

ESMF_GRID_HORZ_STAGGER_B_NE Arakawa B staggering where both the U and V velocities are located at each cell's NorthEast corner. A grid created with this staggering will only accept `ESMF_Fields` with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NECORNER

ESMF_GRID_HORZ_STAGGER_B_NW Arakawa B staggering where both the U and V velocities are located at each cell's NorthWest corner. A grid created with this staggering will only accept `ESMF_Fields` with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NWCORNER

ESMF_GRID_HORZ_STAGGER_B_SE Arakawa B staggering where both the U and V velocities are located at each cell's SouthEast corner. A grid created with this staggering will only accept `ESMF_Fields` with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SECORNER

ESMF_GRID_HORZ_STAGGER_B_SW Arakawa B staggering where both the U and V velocities are located at each cell's SouthWest corner. A grid created with this staggering will only accept `ESMF_Fields` with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SWCORNER

ESMF_GRID_HORZ_STAGGER_C_NE Arakawa C staggering where the U velocity is located at the East face and the V velocity is located at the North face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NFACE ESMF_CELL_EFACE

ESMF_GRID_HORZ_STAGGER_C_NW Arakawa C staggering where the U velocity is located at the West face and the V velocity is located at the North face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NFACE ESMF_CELL_WFACE

ESMF_GRID_HORZ_STAGGER_C_SE Arakawa C staggering where the U velocity is located at the East face and the V velocity is located at the South face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SFACE ESMF_CELL_EFACE

ESMF_GRID_HORZ_STAGGER_C_SW Arakawa C staggering where the U velocity is located at the West face and the V velocity is located at the South face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SFACE ESMF_CELL_WFACE

ESMF_GRID_HORZ_STAGGER_D_NE Arakawa D staggering where the U velocity is located at the North face and the V velocity is located at the East face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NFACE ESMF_CELL_EFACE

ESMF_GRID_HORZ_STAGGER_D_NW Arakawa D staggering where the U velocity is located at the North face and the V velocity is located at the West face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_NFACE ESMF_CELL_WFACE

ESMF_GRID_HORZ_STAGGER_D_SE Arakawa D staggering where the U velocity is located at the South face and the V velocity is located at the East face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SFACE ESMF_CELL_EFACE

ESMF_GRID_HORZ_STAGGER_D_SW Arakawa D staggering where the U velocity is located at the South face and the V velocity is located at the West face. A grid created with this staggering will only accept ESMF_Fields with the following horzRellocs (horizontal relative locations): ESMF_CELL_CENTER ESMF_CELL_SFACE ESMF_CELL_WFACE

ESMF_GRID_HORZ_STAGGER_UNKNOWN Unknown or undefined staggering.

25.2.5 ESMF_GridType

DESCRIPTION:
Grid types supported by ESMF. In general, we expect each ESMF_GridType to have its own explicit Create function. Valid values are:

ESMF_GRID_TYPE_LATLON Latitude/longitude grid

ESMF_GRID_TYPE_LATLON_UNI Uniform latitude/longitude grid

ESMF_GRID_TYPE_UNKNOWN Unknown or undefined grid.

ESMF_GRID_TYPE_XY XY cartesian grid with variable or unequal spacing

ESMF_GRID_TYPE_XY_UNI XY cartesian grid with equal spacing

25.2.6 ESMF_GridVertStagger

DESCRIPTION:
Vertical Grid staggerings supported by ESMF.
Valid values are:

ESMF_GRID_VERT_STAGGER_BOTTOM Vertical velocity or pressure gradient is located at the bottom vertical face of the cell. A grid created with this staggering will only accept ESMF_Fields with the following vertRellocs (vertical relative locations): ESMF_CELL_CELL ESMF_CELL_BOTFACE

ESMF_GRID_VERT_STAGGER_CENTER Vertical velocity or pressure gradient is located at vertical midpoints. A grid created with this staggering will only accept ESMF_Fields with the following vertRellocs (vertical relative locations): ESMF_CELL_CELL

ESMF_GRID_VERT_STAGGER_TOP Vertical velocity or pressure gradient is located at the top vertical face of the cell. A grid created with this staggering will only accept ESMF_Fields with the following vertRellocs (vertical relative locations): ESMF_CELL_CELL ESMF_CELL_TOPFACE

ESMF_GRID_VERT_STAGGER_UNKNOWN Unknown or undefined staggering.

25.2.7 ESMF_GridVertType

DESCRIPTION:
Vertical grid types supported by ESMF. In general, we expect each ESMF_GridVertType to have its own explicit Add subroutine.
Valid values are:

ESMF_GRID_TYPE_VERT_HEIGHT Vertical grid with zero coordinate at bottom

ESMF_GRID_TYPE_VERT_UNKNOWN Unknown or undefined vertical grid

25.3 Use and Examples

In typical applications, Grids are created either internally or read in from a file. The ESMF_Grid class will provide methods for both, though currently it only has routines for simple internal Grid generation. It also has a variety of methods to set and get Grid parameters like the number of cells associated with a particular DE.
The creation of a distributed Grid requires multiple steps, as illustrated in the example code below. The ESMF_GridCreateHorz<Gri call, which has an explicit interface for each GridType, allocates space for the Grid class and sets parameters defining the horizontal grid. A vertical subGrid can then be attached to the Grid via an ESMF_GridAddVert<VertGridType>() call. Currently a Grid can have only a single vertical subGrid. The last call, ESMF_GridDistribute(), allocates some of the Grid subclasses and distributes the Grid in either a default or user-specified decomposition. Currently, these calls must be made in this order (i.e. it is not possible to add a vertical subGrid to an already distributed Grid).

```
! !PROGRAM: ESMF_GridCreateEx - Grid creation
!
! !DESCRIPTION:
!
! This program shows examples of different methods to create 2D and 3D grids
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate two grids
      type(ESMF_Grid) :: grid1, grid2

      ! instantiate horizontal and vertical grid staggerings
```

```
type(ESMF_GridHorzStagger) :: horz_stagger
type(ESMF_GridVertStagger) :: vert_stagger

! local variables for Create routines
integer :: counts(2), countsPerDE1(2), countsPerDE2(2)
character(len=ESMF_MAXSTR) :: name
real(ESMF_KIND_R8), dimension(2) :: min, max
real(ESMF_KIND_R8) :: delta1(40), delta2(50), delta3(10)

! return code
integer :: rc


! initialize ESMF framework
call ESMF_Initialize(rc=rc)
```

25.3.1 Uniform 2D Grid Creation

This example shows how to create a simple uniform horizontal ESMF_Grid.

```
! set the global number of computational cells in each direction
counts(1)    = 10
counts(2)    = 12

! set the global coordinate extrema
min(1)       = 0.0
max(1)       = 10.0
min(2)       = 0.0
max(2)       = 12.0

! set the staggering for the horizontal grid
horz_stagger = ESMF_GRID_HORZ_STAGGER_A

! and add a name to the grid
name         = "test grid 1"

! create a 2 x 2 layout for the Grid
layout = ESMF_DELayoutCreate(vm, (/ 2, 2 /), rc=rc)


! initialize the grid with the above values
grid1 = ESMF_GridCreateHorzXYUni(counts=counts, &
                                 minGlobalCoordPerDim=min, &
                                 maxGlobalCoordPerDim=max, &
                                 horzstagger=horz_stagger, &
                                 name=name, rc=rc)

! distribute the grid
call ESMF_GridDistribute(grid1, delayout=layout, rc=rc)



print *, "Grid example 1 returned"
```

```
call ESMF_GridDestroy(grid1, rc)

print *, "Grid example 1 destroyed"
```

25.3.2 3D Grid Creation

This example shows how to create a 3D ESMF_Grid with specified, non-uniform spacing.

```
! initialize the grid with the above values
grid2 = ESMF_GridCreateHorzLatLon(minGlobalCoordPerDim=min, &
                                  delta1=delta1, delta2=delta2, &
                                  horzstagger=horz_stagger, &
                                  name=name, rc=rc)

! add a vertical subgrid to the horizontal grid
! note: the vertical subgrid must be added before the grid is
!      distributed
call ESMF_GridAddVertHeight(grid2, delta3, vertstagger=vert_stagger, &
                            rc=rc)



! distribute the grid using the same layout as from the first example
! but specifying the decomposition of computational cells
call ESMF_GridDistribute(grid2, delayout=layout, &
                         countsPerDEDim1=countsPerDE1, &
                         countsPerDEDim2=countsPerDE2, &
                         rc=rc)



print *, "Grid example 2 returned"

call ESMF_GridDestroy(grid2, rc)

print *, "Grid example 2 destroyed"


call ESMF_Finalize(rc)


end program ESMF_GridCreateEx
```

25.4 Restrictions and Future Work

1. **Support is limited to 3D, logically rectangular grids.** Currently the only interfaces supported are for three-dimensional, logically rectangular grids.
2. **Support is limited to 2D grid distributions.** The decomposition of grids is limited to two dimensions.
3. **Future Grid Create methods.** Currently grids can only be created by internal generation. In the future, the following create methods will be added:

   ESMF_GridCreateCopy Create a new Grid by copying another Grid

   ESMF_GridCreateCutout Create a new Grid as a subset of an existing Grid

ESMF_GridCreateDiffRes Create a new Grid by coarsening or refining an existing Grid

ESMF_GridCreateExchange Create a new Grid from the intersection of two existing grids

ESMF_GridCreateRead Create a new Grid by reading in from a file

4. **Future Grid types.** The following grids will be supported, although only some will have internal generation methods:

   ESMF_GRID_TYPE_CART_SPECT Spectral space for cartesian coordinates

   ESMF_GRID_TYPE_CUBEDSPHERE Cubed sphere grid

   ESMF_GRID_TYPE_DATASTREAM Data stream - set of locations

   ESMF_GRID_TYPE_DIPOLE Displaced-pole dipole grid

   ESMF_GRID_TYPE_EXCHANGE Intersection of two grids, which is itself a grid

   ESMF_GRID_TYPE_GEODESIC Spherical geodesic grid

   ESMF_GRID_TYPE_LATLON_GAUSS Latitude/Longitude grid with gaussian-spaced latitudes

   ESMF_GRID_TYPE_LATLON_MERC Latitude/Longitude grid with Mercator-spaced latitudes

   ESMF_GRID_TYPE_PHYSFOURIER Mixed Fourier Space/Physcal Space grid

   ESMF_GRID_TYPE_REDUCED Latitude/Longitude grid where the number of longitudinal points is a function of the latitude

   ESMF_GRID_TYPE_SPHER_SPECT Spectral space for spherical harmonics

   ESMF_GRID_TYPE_TRIPOLE Tripolar grids

5. **Future coordinate system support.** Support for the following coordinate systems will be added:

   ESMF_COORD_SYSTEM_CYLINDRICAL Cylindrical coordinates

   ESMF_COORD_SYSTEM_LAGRANGIAN Lagrangian coordinates

   ESMF_COORD_SYSTEM_LATFOURIER Mixed latitude/Fourier spectral space

   ESMF_COORD_SYSTEM_SPECTRAL Wavenumber space

   ESMF_COORD_SYSTEM_USER User-defined coordinate system

6. **Future horizontal Grid staggerings.** Support for the following horizontal staggerings will be added:

   ESMF_GRID_HORZ_STAGGER_E Arakawa E

   ESMF_GRID_HORZ_STAGGER_Z C grid equivalent for geodesic grid

7. **Future vertical Grid types.** Support for the following vertical grids will be added:

   ESMF_COORD_SYSTEM_DEPTH Vertical z coordinate depth (0 at top surface)

   ESMF_COORD_SYSTEM_ETA Vertical eta coordinate

   ESMF_COORD_SYSTEM_HYBRID Hybrid vertical coordinates

   ESMF_COORD_SYSTEM_ISOPYCNAL Vertical density coordinate

   ESMF_COORD_SYSTEM_LAGRANGIAN Lagrangian coordinates

   ESMF_COORD_SYSTEM_PRESSURE Vertical pressure coordinate

   ESMF_COORD_SYSTEM_SIGMA Vertical sigma coordinate

   ESMF_COORD_SYSTEM_THETA Vertical theta coordinate

8. **Future Grid masks.** Grid masks will be implemented, including support for the following mask types:

   ESMF_GRID_MASKTYPE_LOGICAL Logical mask

   ESMF_GRID_MASKTYPE_MULT Multiplicative mask

   ESMF_GRID_MASKTYPE_REGION_ID Integer assigning unique ID to each point

25.5 Design and Implementation Notes

Overall Grid design strategy...

25.5.1 Grid Classes

The Grid class contains two internal classes: the DistGrid (Distributed Grid) class and the PhysGrid (Physical Grid) class. The separation into two classes allows the code to differentiate between functions which define the local decomposition of data and the local representation of the grid. The Grid class itself maintains general information about the global grid (e.g. the grid type, staggering, and coordinate system). The Grid class is relatively thin and otherwise presents a unified interface for DistGrid and PhysGrid functions. Each Grid contains at least one DistGrid and one PhysGrid:

- **DistGrid** The DistGrid class maintains the relationship of how a DELayout maps onto a Grid representation and how that Grid is distributed. DistGrids can represent the same Grid but have different mappings (staggerings) and can be contained by the same Grid object. The DistGrid class represents the mapping between the global Grid and the local data distribution; it has methods to aid in the collection and communication of global data.
- **PhysGrid** The PhysGrid class maintains a local physical representation of a Grid, including all necessary data and masks. PhysGrids can represent subGrids, like vertical Grids, of a single Grid and be contained by the same Grid object.

Some methods which have a Grid interface are actually implemented at the underlying DistGrid or PhysGrid level; they will be inherited by the Grid class. This allows the API to present functions at the level which is most consistent to the application without restricting where inside the ESMF the actual implementation is done.

25.5.2 DistGrid Implementation Notes

The DistGrid class contains the mapping between the local grid decompositions and the global logical Grid. It contains methods to synchronize data values between the boundaries of subsets, and to collect and communicate global data values. It interacts closely with the PhysGrid object.

1. **DistGrid Internal Classes**

   The DistGrid class contains the DELayout class as well as two private subclasses, the DistGridGlobal and DistGridLocal classes. The separation between DistGridGlobal and DistGridLocal allows the code to clearly differentiate between functions which operate internal to a single DE on a local decomposition of data, and those which must be aware of the global state of the distribution.

   - **DELayout** The DELayout class is described in detail in the Utilities section of this document.
   - **DistGridGlobal** The DistGridGlobal class contains general information about each of the partitions that the entire grid has been decomposed into. This includes information about how each part relates to the whole, how many points/cells there are per decomposition, etc. This information allows DistGrid to compute information about other decompositions on other PEs without having to do communication first.
   - **DistGridLocal** The DistGridLocal class contains detailed subgrid information for the data located on this PE, such as the local cell count and the number of cells along each axis and their position in the global Grid. When we implement multiple DEs per PE then we will have a list of these instead of a single one in the DistGrid class.

2. **Local verses Global Data**

   The primary purpose of DistGrid is to encapsulate information about the local decomposition(s) (DE) of the Grid on this PE. This includes such information as the total number of local cells, if logically rectangular the numbers of cells along each dimension, and the relative location of this DE compared to the overall ESMF_Grid. The minimum information required would be to compute and store data only for the local DE.

   However, at create time DistGrid computes information not only about the local decomposition, but also less detailed information about the other decompositions for the entire Grid. While this duplicates some data, it avoids communication when a DE requires information to enable it to send data to or receive data from other DEs,

3. **Boundary Cells**

   As part of the create-time computation DistGrid computes sizes and lengths for the local DE grid cells, and also does a secondary computation of sizes and lengths taking into account a layer of boundary cells around each DE. These boundary cells are distinct from the halo cells which are specified on a per-Field basis and are visible to the user code.

   The boundary cells inside DistGrid are only used internally to the Framework, for example during regridding to avoid unnecessary inter-DE communication and to handle exterior boundaries in a consistent manner.

Some methods which have a DistGrid interface will actually be implemented at the underlying DELayout or Array level; they will be inherited by the DistGrid class. This allows the user API (Application Programming Interface) to present functions at the level which is most consistent to the application without restricting where inside the ESMF the actual implementation is done.
The DistGrid class has two instances of both DistGridLocal and DistGridGlobal classes, one to represent the computational domain and one to represent the total domain, which includes halo and ghost cells as well as computational cells.

25.5.3 PhysGrid Implementation Notes

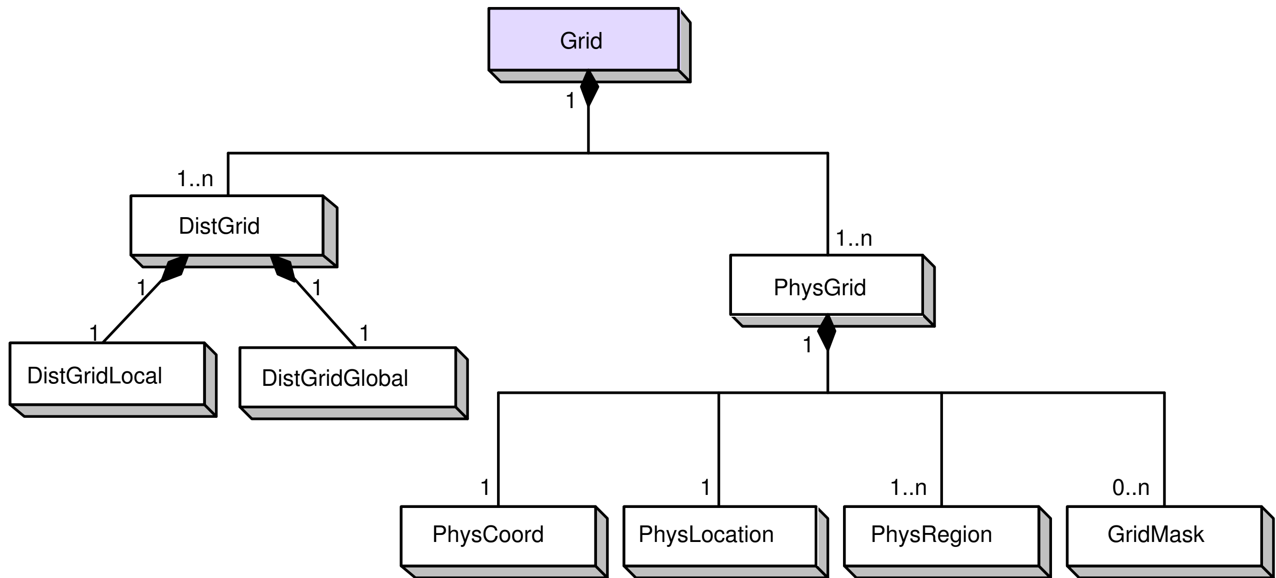
The PhysGrid class is itself private and is part of the Grid class. It is designed to contain all information describing physical properties of the Grid, as well as methods to initialize them and to calculate user-requested metrics.
The PhysGrid class contains the following private classes:

- **ESMF_GridMask** Data type describing masks for a PhysGrid. Masks are named and can be of different types, including logical masks, multiplicative masks, and integer region IDs.
- **ESMF_PhysCoord** A physical coordinate carries information describing coordinate attributes like names and flags for special properties of a coordinate axis. This information is used by PhysGrid and Grid to help describe the complete physical properties of a grid.
- **ESMF_PhysLocation** Physical locations for a set of points defining the grid.
- **ESMF_PhysRegion** Physical locations for a set of points defining regions of the grid (e.g. cell vertices or domains of influence).

There is a correspondence between the DistGrid class and the PhysGrid class. The PhysGrid class maintains all the local data necessary to represent the Grid, while the DistGrid class describes the local extents of that data and its relationship to the global decomposition. Together, a PhysGrid and related DistGrid define a representation of a Grid. There is a correspondence between the PhysGrid class and the Field class as well: the PhysGrid data on a DE describes the physical location of the corresponding Field data.
The PhysGrid class maintains a local physical representation of a Grid, including all necessary data and masks. PhysGrids can represent subGrids of a single Grid and be contained by the same Grid object. The PhysGrid class must have methods that can internally generate a variety of computational grids in a distributed environment from relatively simple input. The PhysGrid data has to be accessible to the ESMF user in a variety of specified ways or metrics, and it must have the capability to attach a number of masks or identifiers. Please note that the PhysGrid class is designed to be an private class; all access to its contents are via Grid methods.

25.6 Object Model

The following is a simplified UML diagram showing the structure of the Grid class. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



Each Grid contains a Distributed Grid and a Physical Grid. The Physical Grid maintains information about the global coordinates. In general this data is described implicitly by specifying the grid type and the corresponding parameters. However it is possible that the Physical Grid must be completely enumerated, perhaps in the case of assimilated data or unstructured data. The Distributed Grid defines an index space that corresponds to cells in the Physical Grid and is decomposed among DEs in a DELayout.

25.7 Class API: General Grid Methods

25.7.1 ESMF_GridAddVertHeight - Add a vertical dimension to an existing Grid

INTERFACE:

```
subroutine ESMF_GridAddVertHeight(grid, delta, coord, vertstagger, &
                                  dimName, dimUnit, name, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: delta
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: coord
type(ESMF_GridVertStagger), intent(in), optional :: vertstagger
character(len=*), intent(in), optional :: dimName
character(len=*), intent(in), optional :: dimUnit
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

This routine adds a vertical subGrid to an already allocated `grid`. This explicit interface only creates vertical subGrids with coordinate systems where the zero point is defined at the bottom. Only one vertical subGrid is allowed for any `ESMF_Grid`, so if a vertical subGrid already exists for the Grid that is passed in, an error is returned. This routine generates `ESMF_Grid` coordinates from either of two optional sets of arguments:

1. given array of deltas (variable delta), assuming 0 is the minimum or starting coordinate
2. given array of coordinates (variable coords)

If neither of these sets of arguments is present and valid, an error message is issued and an error code returned.
The arguments are:

grid `ESMF_Grid` to add vertical grid to.

[delta] Array of physical increments in the vertical direction.

[coord] Array of physical coordinates in the vertical direction.

[vertstagger] `ESMF_GridVertStagger` specifier to denote vertical grid stagger.

[dimName] Dimension name.

[dimUnit] Dimension unit.

[name] Name for the vertical grid.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.2 ESMF_GridCreate - Create a new Grid with no contents

INTERFACE:

```
! Private name; call using ESMF_GridCreate()
function ESMF_GridCreateEmpty(name, rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridCreateEmpty
```

ARGUMENTS:

```
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_Grid` object and constructs its internals, but does not fill in any contents. Returns a pointer to the new `ESMF_Grid`.
The arguments are:

[name] `ESMF_Grid` name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.3 ESMF_GridDestroy - Free all resources associated with a Grid

INTERFACE:

```
subroutine ESMF_GridDestroy(grid, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
integer, intent(out), optional :: rc
```

DESCRIPTION:

Destroys an `ESMF_Grid` object previously allocated via an `ESMF_GridCreate routine`.
The arguments are:

grid `ESMF_Grid` to be destroyed.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

!REQUIREMENTS:

25.7.4 ESMF_GridDistribute - Distribute a Grid that has already been initialized

INTERFACE:

```
subroutine ESMF_GridDistribute(grid, delayout, countsPerDEDim1, &
                               countsPerDEDim2, decompIds, name, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_DELayout), intent(in) :: delayout
integer, dimension(:), intent(in), optional :: countsPerDEDim1
integer, dimension(:), intent(in), optional :: countsPerDEDim2
integer, dimension(:), intent(in), optional :: decompIds
character (len = *), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the decomposition of the `grid`.
The arguments are:

grid `ESMF_Grid` to be distributed.

delayout `ESMF_DELayout` on which the `grid` is to be decomposed.

[countsPerDEDim1] Array of number of grid increments per DE in the x-direction.

[countsPerDEDim2] Array of number of grid increments per DE in the y-direction.

[decompIds] Identifier for which `grid` axes are decomposed.

[name] `ESMF_Grid` name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

!REQUIREMENTS:

25.7.5 ESMF_GridGet - Get a variety of information about a Grid

INTERFACE:

```
subroutine ESMF_GridGet(grid, horzrelloc, vertrelloc, &
                        horzgridtype, vertgridtype, &
                        horzstagger, vertstagger, &
                        horzcoordsystem, vertcoordsystem, &
                        coordorder, dimCount, minGlobalCoordPerDim, &
                        maxGlobalCoordPerDim, globalCellCountPerDim, &
                        globalStartPerDEPerDim, maxLocalCellCountPerDim, &
                        cellCountPerDEPerDim, periodic, delayout, &
                        name, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_RelLoc), intent(in), optional :: horzrelloc
type(ESMF_RelLoc), intent(in), optional :: vertrelloc
type(ESMF_GridType),     intent(out), optional :: horzgridtype
type(ESMF_GridVertType), intent(out), optional :: vertgridtype
type(ESMF_GridHorzStagger), intent(out), optional :: horzstagger
type(ESMF_GridVertStagger), intent(out), optional :: vertstagger
type(ESMF_CoordSystem), intent(out), optional :: horzcoordsystem
type(ESMF_CoordSystem), intent(out), optional :: vertcoordsystem
type(ESMF_CoordOrder),  intent(out), optional :: coordorder
integer, intent(out), optional :: dimCount
real(ESMF_KIND_R8), intent(out), dimension(:), optional :: &
                      minGlobalCoordPerDim
real(ESMF_KIND_R8), intent(out), dimension(:), optional :: &
                      maxGlobalCoordPerDim
integer, intent(out), dimension(:), optional :: globalCellCountPerDim
integer, intent(out), dimension(:,:), optional :: globalStartPerDEPerDim
integer, intent(out), dimension(:), optional :: maxLocalCellCountPerDim
integer, intent(out), dimension(:,:), optional :: cellCountPerDEPerDim
type(ESMF_Logical), intent(out), dimension(:), optional :: periodic
type(ESMF_DELayout), intent(out), optional :: delayout
character(len = *), intent(out), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Gets information about an `ESMF_Grid`, depending on a list of optional arguments.
The arguments are:

grid `ESMF_Grid` to be queried.

[horzrelloc] `ESMF_RelLoc` identifier corresponding to the horizontal grid.

[vertrelloc] `ESMF_RelLoc` identifier corresponding to the vertical grid.

[horzgridtype] `ESMF_GridType` specifier to denote horizontal grid type.

[vertgridtype] `ESMF_GridVertType` specifier to denote vertical grid type.

[horzstagger] `ESMF_GridHorzStagger` specifier to denote horizontal grid stagger.

[vertstagger] `ESMF_GridHorzStagger` specifier to denote vertical grid stagger.

[horzcoordsystem] ESMF_CoordSystem which identifies an ESMF standard coordinate system (e.g. spherical, cartesian, pressure, etc.) for the horizontal grid.

[vertcoordsystem] ESMF_CoordSystem which identifies an ESMF standard coordinate system (e.g. spherical, cartesian, pressure, etc.) for the vertical grid.

[coordorder] ESMF_CoordOrder specifier to denote coordinate ordering.

[dimCount] Number of dimensions represented by this grid.

[minGlobalCoordPerDim] Array of minimum global physical coordinates in each direction.

[maxGlobalCoordPerDim] Array of maximum global physical coordinates in each direction.

[globalCellCountPerDim] Array of numbers of global grid increments in each direction.

[globalStartPerDEPerDim] Array of global starting locations for each DE and in each direction.

[maxLocalCellCountPerDim] Array of maximum grid counts on any DE in each direction.

[cellCountPerDEPerDim] 2-D array of grid counts on each DE and in each direction.

[periodic] Returns the periodicity along the coordinate axes - logical array.

[name] ESMF_Grid name.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

!REQUIREMENTS:

25.7.6 ESMF_GridGetAttribute - Retrieve a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetInt4Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer attribute from the grid.
The arguments are:

grid An ESMF_Grid object.

name The name of the attribute to retrieve.

value The 4-byte integer value of the named attribute.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

25.7.7 ESMF_GridGetAttribute - Retrieve a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetInt4ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte integer list attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 4-byte integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.8 ESMF_GridGetAttribute - Retrieve an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetInt8Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

value The 8-byte integer value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.9 ESMF_GridGetAttribute - Retrieve an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetInt8ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte integer list attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 8-byte integer values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.10 ESMF_GridGetAttribute - Retrieve a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetReal4Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

value The 4-byte real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.11 ESMF_GridGetAttribute - Retrieve a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetReal4ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a 4-byte real list attribute from an `ESMF_Grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The 4-byte real values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.12 ESMF_GridGetAttribute - Retrieve an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetReal8Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte real attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

value The 8-byte real value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.13 ESMF_GridGetAttribute - Retrieve an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetReal8ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns an 8-byte real list attribute from an `ESMF_Grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The real*8 values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.14 ESMF_GridGetAttribute - Retrieve a logical attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetLogicalAttr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

value The logical value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.15 ESMF_GridGetAttribute - Retrieve a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetLogicalListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(out) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a logical list attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

count The number of values in the attribute.

valueList The logical values of the named attribute. The list must be at least `count` items long.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.16 ESMF_GridGetAttribute - Retrieve a character attribute

INTERFACE:

```
! Private name; call using ESMF_GridGetAttribute()
subroutine ESMF_GridGetCharAttr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
character (len = *), intent(out) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns a character attribute from the `grid`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to retrieve.

value The character value of the named attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.17 ESMF_GridGetAttributeCount - Query the number of attributes

INTERFACE:

```
subroutine ESMF_GridGetAttributeCount(grid, count, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
integer, intent(out) :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns the number of attributes associated with the given `grid` in the argument `count`.
The arguments are:

grid An `ESMF_Grid` object.

count The number of attributes associated with this object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.18 ESMF_GridGetAttributeInfo - Query Grid attributes by name

INTERFACE:

```
! Private name; call using ESMF_GridGetAttributeInfo()
subroutine ESMF_GridGetAttrInfoByName(grid, name, datatype, &
                                       datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character(len=*), intent(in) :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the named attribute, including `datatype`, `datakind` (if applicable), and `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to query.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

count The number of items in this attribute. For character types, the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.19 ESMF_GridGetAttributeInfo - Query Grid attributes by index number

INTERFACE:

```
! Private name; call using ESMF_GridGetAttributeInfo()
subroutine ESMF_GridGetAttrInfoByNum(grid, attributeIndex, name, &
                                     datatype, datakind, count, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
integer, intent(in) :: attributeIndex
character(len=*), intent(out), optional :: name
type(ESMF_DataType), intent(out), optional :: datatype
type(ESMF_DataKind), intent(out), optional :: datakind
integer, intent(out), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed attribute, including `datatype`, `datakind` (if applicable), and item `count`.
The arguments are:

grid An `ESMF_Grid` object.

attributeIndex The index number of the attribute to query.

name Returns the name of the attribute.

[datatype] The data type of the attribute. One of the values `ESMF_DATA_INTEGER`, `ESMF_DATA_REAL`, `ESMF_DATA_LOGICAL`, or `ESMF_DATA_CHARACTER`.

[datakind] The datakind of the attribute, if attribute is type `ESMF_DATA_INTEGER` or `ESMF_DATA_REAL`. One of the values `ESMF_I4`, `ESMF_I8`, `ESMF_R4`, or `ESMF_R8`. For all other types the value `ESMF_NOKIND` is returned.

count Returns the number of items in this attribute. For character types, this is the length of the character string.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.20 ESMF_GridGetCoord - Get the coordinates of a Grid

INTERFACE:

```
subroutine ESMF_GridGetCoord(grid, horzrelloc, vertrelloc, centerCoord, &
                             cornerCoord, faceCoord, reorder, total, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_RelLoc), intent(in), optional :: horzrelloc
type(ESMF_RelLoc), intent(in), optional :: vertrelloc
type(ESMF_Array), intent(out), dimension(:), optional :: centerCoord
type(ESMF_Array), intent(out), dimension(:), optional :: cornerCoord
type(ESMF_Array), intent(out), optional :: faceCoord
```

```
logical, intent(in), optional :: reorder
logical, intent(in), optional :: total
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns coordinate information for the `grid`.
The arguments are:

grid `ESMF_Grid` to be queried.

[horzrelloc] Horizontal relative location of the `grid` to be queried.

[vertrelloc] Vertical relative location of the `grid` to be queried.

[centerCoord] Coordinates of each cell center. The dimension index should be defined first (e.g. x = coord(1,i,j), y=coord(2,i,j)).

[cornerCoord] Coordinates of corners of each cell. The dimension index should be defined first, followed by the corner index. Corners can be numbered in either clockwise or counter-clockwise direction, but must be numbered consistently throughout grid.

[faceCoord] Coordinates of corners of each cell. The dimension index should be defined first, followed by the face index. Faces should be numbered consistently with corners. For example, face 1 should correspond to the face between corners 1,2.

[reorder] Logical. If TRUE, reorder any results using the GridOrder before returning. If FALSE do not reorder. The default value is TRUE and users should not need to reset this for most applications.

[total] Logical. If TRUE, return the total coordinates including internally generated boundary cells. If FALSE return the computational cells (which is what the user will be expecting.)

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

!REQUIREMENTS:

25.7.21 ESMF_GridGetDELocalInfo - Get local DE information for a Grid

INTERFACE:

```
subroutine ESMF_GridGetDELocalInfo(grid, horzrelloc, vertrelloc, &
                          myDE, localCellCount, localCellCountPerDim, &
                          minLocalCoordPerDim, maxLocalCoordPerDim, &
                          globalStartPerDim, reorder, total, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_RelLoc), intent(in) :: horzrelloc
type(ESMF_RelLoc), intent(in), optional :: vertrelloc
integer, intent(inout), optional :: myDE
integer, intent(inout), optional :: localCellCount
integer, dimension(:), intent(inout), optional :: localCellCountPerDim
real(ESMF_KIND_R8), intent(out), dimension(:), optional :: &
                     minLocalCoordPerDim
real(ESMF_KIND_R8), intent(out), dimension(:), optional :: &
                     maxLocalCoordPerDim
```

```
integer, dimension(:), intent(inout), optional :: globalStartPerDim
logical, intent(in), optional :: reorder
logical, intent(in), optional :: total
integer, intent(out), optional :: rc
```

DESCRIPTION:

Gets grid information for a particular Decomposition Element (DE).
The arguments are:

grid ESMF_Grid to be queried.

horzrelloc ESMF_RelLoc identifier corresponding to the horizontal grid.

[vertrelloc] ESMF_RelLoc identifier corresponding to the vertical grid.

[myDE] Identifier for this ESMF_DE, zero-based.

[localCellCount] Local (on this ESMF_DE) number of cells.

[localCellCountPerDim] Local (on this ESMF_DE) number of cells per dimension.

[minLocalCoordPerDim] Array of minimum local physical coordinates in each dimension.

[maxLocalCoordPerDim] Array of maximum local physical coordinates in each dimension.

[globalStartPerDim] Global index of starting counts for each dimension.

[reorder] Logical. If TRUE, reorder any results using the GridOrder before returning. If FALSE do not reorder. The default value is TRUE and users should not need to reset this for most applications.

[total] Logical flag to indicate getting DistGrid information for total cells. The default is the computational regime.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

!REQUIREMENTS:

25.7.22 ESMF_GridGlobalToDELocalIndex - Translate global indexing to DE local

INTERFACE:

```
subroutine ESMF_GridGlobalToDELocalIndex(grid, horzrelloc, vertrelloc, &
                                         global1D, local1D, &
                                         global2D, local2D, &
                                         dimOrder, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_RelLoc), intent(in) :: horzrelloc
type(ESMF_RelLoc), intent(in), optional :: vertrelloc
integer(ESMF_KIND_I4), dimension(:), optional, intent(in) :: global1D
integer(ESMF_KIND_I4), dimension(:), optional, intent(out) :: local1D
integer(ESMF_KIND_I4), dimension(:,:), optional, intent(in) :: global2D
integer(ESMF_KIND_I4), dimension(:,:), optional, intent(out) :: local2D
integer, dimension(:), optional, intent(in) :: dimOrder
integer, intent(out), optional :: rc
```

DESCRIPTION:

Translates an array of integer cell identifiers from global indexing to DE-local indexing.
The arguments are:

grid ESMF_Grid to be used.

horzrelloc ESMF_RelLoc identifier corresponding to the horizontal grid.

[vertrelloc] ESMF_RelLoc identifier corresponding to the vertical grid.

[global1D] One-dimensional array of global identifiers to be translated. Infers translating between positions in memory.

[local1D] One-dimensional array of local identifiers corresponding to global identifiers.

[global2D] Two-dimensional array of global identifiers to be translated. Infers translating between indices in ij space.

[local2D] Two-dimensional array of local identifiers corresponding to global identifiers.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

!REQUIREMENTS:

25.7.23 ESMF_GridDELocalToGlobalIndex - Translate DE local indexing to global

INTERFACE:

```
subroutine ESMF_GridDELocalToGlobalIndex(grid, horzrelloc, vertrelloc, &
                                         local1D, global1D, &
                                         local2D, global2D, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_RelLoc), intent(in) :: horzrelloc
type(ESMF_RelLoc), intent(in), optional :: vertrelloc
integer(ESMF_KIND_I4), dimension(:), optional, intent(in) ::  local1D
integer(ESMF_KIND_I4), dimension(:), optional, intent(out) :: global1D
integer(ESMF_KIND_I4), dimension(:,:), optional, intent(in) ::  local2D
integer(ESMF_KIND_I4), dimension(:,:), optional, intent(out) :: global2D
integer, intent(out), optional :: rc
```

DESCRIPTION:

Translates an array of integer cell identifiers from DE-local indexing to global indexing.
The arguments are:

grid ESMF_Grid to be used.

horzrelloc ESMF_RelLoc identifier corresponding to the horizontal grid.

[vertrelloc] ESMF_RelLoc identifier corresponding to the vertical grid.

[local1D] One-dimensional array of local identifiers to be translated. Infers translating between positions in memory.

[global1D] One-dimensional array of global identifiers corresponding to local identifiers.

[local2D] Two-dimensional array of local identifiers to be translated. Infers translating between indices in ij space.

[global2D] Two-dimensional array of global identifiers corresponding to local identifiers.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

!REQUIREMENTS:

25.7.24 ESMF_GridPrint - Print the contents of a Grid

INTERFACE:

```
subroutine ESMF_GridPrint(grid, options, rc)
```

ARGUMENTS:

```
type(ESMF_Grid),   intent(in) :: grid
character (len=*), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `grid` to `stdout`.
The arguments are:

grid `ESMF_Grid` to print.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.25 ESMF_GridSet - Set a variety of information about a Grid

INTERFACE:

```
subroutine ESMF_GridSet(grid, horzgridtype, vertgridtype, &
                        horzstagger, vertstagger, &
                        horzcoordsystem, vertcoordsystem, &
                        coordorder, minGlobalCoordPerDim, &
                        maxGlobalCoordPerDim, periodic, name, rc)
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid
type(ESMF_GridType),     intent(in), optional :: horzgridtype
type(ESMF_GridVertType), intent(in), optional :: vertgridtype
type(ESMF_GridHorzStagger), intent(in), optional :: horzstagger
type(ESMF_GridVertStagger), intent(in), optional :: vertstagger
type(ESMF_CoordSystem), intent(in), optional :: horzcoordsystem
type(ESMF_CoordSystem), intent(in), optional :: vertcoordsystem
type(ESMF_CoordOrder), intent(in), optional :: coordorder
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: minGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: maxGlobalCoordPerDim
type(ESMF_Logical), intent(in), optional :: periodic(:)
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets information for the grid.
The arguments are:

grid ESMF_Grid to be modified.

[horzgridType] ESMF_GridType specifier to denote horizontal grid type.

[vertgridType] ESMF_GridVertType specifier to denote vertical grid type.

[horzstagger] ESMF_GridHorzStagger specifier to denote horizontal grid stagger.

[vertstagger] ESMF_GridVertStagger specifier to denote vertical grid stagger.

[horzcoordsystem] ESMF_CoordSystem which identifies an ESMF standard coordinate system (e.g. spherical, cartesian, pressure, etc.) for the horizontal grid.

[vertcoordsystem] ESMF_CoordSystem which identifies an ESMF standard coordinate system (e.g. spherical, cartesian, pressure, etc.) for the vertical grid.

[coordorder] ESMF_CoordOrder specifier to denote coordinate ordering.

[minGlobalCoordPerDim] Array of minimum global physical coordinates in each direction.

[maxGlobalCoordPerDim] Array of maximum global physical coordinates in each direction.

[periodic] Logical specifier (array) to denote periodicity along the coordinate axes.

[name] Character string name of ESMF_Grid.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

!REQUIREMENTS:

25.7.26 ESMF_GridSetAttribute - Set a 4-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetInt4Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(inout) :: grid
character (len = *), intent(in) :: name
integer(ESMF_KIND_I4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer attribute to the grid. The attribute has a name and a value.
The arguments are:

grid An ESMF_Grid object.

name The name of the attribute to add.

value The 4-byte integer value of the attribute to add.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

25.7.27 ESMF_GridSetAttribute - Set a 4-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetInt4ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte integer list attribute to the `grid`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The 4-byte integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.28 ESMF_GridSetAttribute - Set an 8-byte integer attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetInt8Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(inout) :: grid
character (len = *), intent(in) :: name
integer(ESMF_KIND_I8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte integer attribute to the `grid`. The attribute has a `name` and a `value`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

value The 8-byte integer value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.29 ESMF_GridSetAttribute - Set an 8-byte integer list attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetInt8ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 8-byte integer list attribute to the `grid`. The attribute has a `name` and a `valueList`. The number of integer items in the `valueList` is given by `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

count The number of integers in the `valueList`.

valueList The 8-byte integer values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.30 ESMF_GridSetAttribute - Set a 4-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetReal4Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
real(ESMF_KIND_R4), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real attribute to the `grid`. The attribute has a `name` and a `value`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

value The 4-byte real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.31 ESMF_GridSetAttribute - Set a 4-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetReal4ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R4), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a 4-byte real list attribute to the `grid`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The 4-byte real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.32 ESMF_GridSetAttribute - Set an 8-byte real attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetReal8Attr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
real(ESMF_KIND_R8), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real attribute to the `grid`. The attribute has a `name` and a `value`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

value The 8-byte real value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.33 ESMF_GridSetAttribute - Set an 8-byte real list attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetReal8ListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
real(ESMF_KIND_R8), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches an 8-byte real list attribute to the `grid`. The attribute has a `name` and a `valueList`. The number of real items in the `valueList` is given by `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

count The number of reals in the `valueList`.

value The 8-byte real values of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.34 ESMF_GridSetAttribute - Set a logical attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetLogicalAttr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
type(ESMF_Logical), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical attribute to the `grid`. The attribute has a `name` and a `value`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

value The logical true/false value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.35 ESMF_GridSetAttribute - Set a logical list attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetLogicalListAttr(grid, name, count, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
integer, intent(in) :: count
type(ESMF_Logical), dimension(:), intent(in) :: valueList
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a logical list attribute to the `grid`. The attribute has a `name` and a `valueList`. The number of logical items in the `valueList` is given by `count`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

count The number of logicals in the `valueList`.

value The logical true/false values of the attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.36 ESMF_GridSetAttribute - Set a character attribute

INTERFACE:

```
! Private name; call using ESMF_GridSetAttribute()
subroutine ESMF_GridSetCharAttr(grid, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len = *), intent(in) :: name
character (len = *), intent(in) :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attaches a character attribute to the `grid`. The attribute has a `name` and a `value`.
The arguments are:

grid An `ESMF_Grid` object.

name The name of the attribute to add.

value The character value of the attribute to add.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.7.37 ESMF_GridValidate - Check validity of a Grid

INTERFACE:

```
subroutine ESMF_GridValidate(grid, options, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
character (len=*), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that an `ESMF_Grid` is internally consistent. Currently checks to make sure: the pointer to the grid is associated; the grid status indicates the grid is ready to use.
The arguments are:

grid `ESMF_Grid` to be validated.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.8 Class API: Logically Rectangular Grid Methods

25.8.1 ESMF_GridCreateHorzLatLon - Create a new horizontal LatLon Grid

INTERFACE:

```
function ESMF_GridCreateHorzLatLon(minGlobalCoordPerDim, &
                                   delta1, delta2, coord1, coord2, &
                                   horzstagger, dimNames, dimUnits, &
                                   coordorder, coordindex, periodic, &
                                   name, rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridCreateHorzLatLon
```

ARGUMENTS:

```
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: minGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: delta1
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: delta2
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: coord1
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: coord2
type(ESMF_GridHorzStagger), intent(in), optional :: horzstagger
character(len=*), dimension(:), intent(in), optional :: dimNames
character(len=*), dimension(:), intent(in), optional :: dimUnits
type(ESMF_CoordOrder), intent(in), optional :: coordorder
type(ESMF_CoordIndex), intent(in), optional :: coordindex
type(ESMF_Logical), dimension(:), intent(in), optional :: periodic
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new ESMF_Grid object, constructs its internals, and internally generates the ESMF_Grid. Returns a pointer to the new ESMF_Grid. This routine creates an ESMF_Grid with the following parameters: logically rectangular; user-specified spacing; horizontal spherical coordinate system. This routine generates ESMF_Grid coordinates from either of two optional sets of arguments: (1). given min and arrays of deltas (variables minGlobalCoordPerDim, delta1 and delta2); (2). given arrays of coordinates (variables coords1 and coords2). If neither of these sets of arguments is present and valid, an error message is issued and the program is terminated.
The arguments are:

[minGlobalCoordsPerDim] Array of minimum physical coordinate in each direction.

[delta1] Array of physical increments between nodes in the first direction.

[delta2] Array of physical increments between nodes in the second direction.

[coord1] Array of physical coordinates in the first direction.

[coord2] Array of physical coordinates in the second direction.

[horzstagger] ESMF_GridHorzStagger specifier to denote horizontal grid stagger.

[dimNames] Array of dimension names.

[dimUnits] Array of dimension units.

[coordorder] ESMF_CoordOrder specifier to denote coordinate ordering.

[coordindex] ESMF_CoordIndex specifier to denote global or local indexing.

[periodic] Logical specifier (array) to denote periodicity along the coordinate axes.

[name] ESMF_Grid name.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

25.8.2 ESMF_GridCreateHorzLatLonUni - Create a new uniform horizontal LatLon Grid

INTERFACE:

```
      function ESMF_GridCreateHorzLatLonUni(counts, minGlobalCoordPerDim, &
                                            maxGlobalCoordPerDim, &
                                            deltaPerDim, horzstagger, &
                                            dimNames, dimUnits, &
                                            coordorder, coordindex, periodic, &
                                            name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateHorzLatLonUni
```

ARGUMENTS:

```
integer, dimension(:), intent(in) :: counts
real(ESMF_KIND_R8), dimension(:), intent(in) :: minGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: &
                                                    maxGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: &
                                                    deltaPerDim
type(ESMF_GridHorzStagger), intent(in), optional :: horzstagger
character(len=*), dimension(:), intent(in), optional :: dimNames
character(len=*), dimension(:), intent(in), optional :: dimUnits
type(ESMF_CoordOrder), intent(in), optional :: coordorder
type(ESMF_CoordIndex), intent(in), optional :: coordindex
type(ESMF_Logical), dimension(:), intent(in), optional :: periodic
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_Grid` object, constructs its internals, and internally generates the `ESMF_Grid`. Returns a pointer to the new `ESMF_Grid`. This routine creates an `ESMF_Grid` with the following parameters: logically rectangular; uniformly spaced coordinates (the distance between any two consecutive grid points is equal); horizontal spherical coordinate system. This routine generates `ESMF_Grid` coordinates from either of two optional sets of arguments: (1). given min, max, and count (variables minGlobalCoordPerDim, maxGlobalCoordPerDim, and counts); (2). given min, delta, and count (variables minGlobalCoordPerDim, deltaPerDim, and counts). If neither of these sets of arguments is present and valid, an error message is issued and the program is terminated.
The arguments are:

counts Array of number of grid increments in each dimension. This array must have at least a length of two and have valid values in the first two array locations or a fatal error occurs.

minGlobalCoordPerDim Array of minimum physical coordinates in each dimension.

[maxGlobalCoordPerDim] Array of maximum physical coordinates in each direction.

[deltaPerDim] Array of constant physical increments in each direction.

[horzstagger] `ESMF_GridHorzStagger` specifier to denote horizontal grid stagger.

[dimNames] Array of dimension names.

[dimUnits] Array of dimension units.

[coordorder] `ESMF_CoordOrder` specifier to denote coordinate ordering.

[coordindex] `ESMF_CoordIndex` specifier to denote global or local indexing.

[periodic] Logical specifier (array) to denote periodicity along the coordinate axes.

[name] `ESMF_Grid` name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.8.3 ESMF_GridCreateHorzXY - Create a new horizontal XY Grid

INTERFACE:

```
function ESMF_GridCreateHorzXY(minGlobalCoordPerDim, &
                               delta1, delta2, coord1, coord2, &
                               horzstagger, dimNames, dimUnits, &
                               coordorder, coordindex, periodic, &
                               name, rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridCreateHorzXY
```

ARGUMENTS:

```
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: minGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: delta1
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: delta2
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: coord1
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: coord2
type(ESMF_GridHorzStagger), intent(in), optional :: horzstagger
character(len=*), dimension(:), intent(in), optional :: dimNames
character(len=*), dimension(:), intent(in), optional :: dimUnits
type(ESMF_CoordOrder), intent(in), optional :: coordorder
type(ESMF_CoordIndex), intent(in), optional :: coordindex
type(ESMF_Logical), dimension(:), intent(in), optional :: periodic
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_Grid` object, constructs its internals, and internally generates the `ESMF_Grid`. Returns a pointer to the new `ESMF_Grid`. This routine creates an `ESMF_Grid` with the following parameters: logically rectangular; user-specified spacing; horizontal cartesian coordinate system. This routine generates `ESMF_Grid` coordinates from either of two optional sets of arguments: (1). given min and arrays of deltas (variables minGlobalCoordPerDim, delta1 and delta2); (2). given arrays of coordinates (variables coords1 and coords2). If neither of these sets of arguments is present and valid, an error message is issued and the program is terminated.
The arguments are:

[minGlobalCoordsPerDim] Array of minimum physical coordinate in each direction.

[delta1] Array of physical increments between nodes in the first direction.

[delta2] Array of physical increments between nodes in the second direction.

[coord1] Array of physical coordinates in the first direction.

[coord2] Array of physical coordinates in the second direction.

[horzstagger] `ESMF_GridHorzStagger` specifier to denote horizontal grid stagger.

[dimNames] Array of dimension names.

[dimUnits] Array of dimension units.

[coordorder] `ESMF_CoordOrder` specifier to denote coordinate ordering.

[coordindex] `ESMF_CoordIndex` specifier to denote global or local indexing.

[periodic] Logical specifier (array) to denote periodicity along the coordinate axes.

[name] `ESMF_Grid` name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

25.8.4 ESMF_GridCreateHorzXYUni - Create a new uniform horizontal XY Grid

INTERFACE:

```
function ESMF_GridCreateHorzXYUni(counts, minGlobalCoordPerDim, &
                                  maxGlobalCoordPerDim, &
                                  deltaPerDim, horzstagger, &
                                  dimNames, dimUnits, &
                                  coordorder, coordindex, periodic, &
                                  name, rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridCreateHorzXYUni
```

ARGUMENTS:

```
integer, dimension(:), intent(in) :: counts
real(ESMF_KIND_R8), dimension(:), intent(in) :: minGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: &
                                                 maxGlobalCoordPerDim
real(ESMF_KIND_R8), dimension(:), intent(in), optional :: &
                                                 deltaPerDim
type(ESMF_GridHorzStagger), intent(in), optional :: horzstagger
character(len=*), dimension(:), intent(in), optional :: dimNames
character(len=*), dimension(:), intent(in), optional :: dimUnits
type(ESMF_CoordOrder), intent(in), optional :: coordorder
type(ESMF_CoordIndex), intent(in), optional :: coordindex
type(ESMF_Logical), dimension(:), intent(in), optional :: periodic
character(len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_Grid` object, constructs its internals, and internally generates the `ESMF_Grid`. Returns a pointer to the new `ESMF_Grid`. This routine creates an `ESMF_Grid` with the following parameters: logically rectangular; uniformly spaced coordinates (the distance between any two consecutive grid points is equal); horizontal cartesian coordinate system. This routine generates `ESMF_Grid` coordinates from either of two optional sets of arguments: (1). given min, max, and count (variables minGlobalCoordPerDim, maxGlobalCoordPerDim, and counts); (2). given min, delta, and count (variables minGlobalCoordPerDim, deltaPerDim, and counts). If neither of these sets of arguments is present and valid, an error message is issued and the program is terminated.
The arguments are:

counts Array of number of grid increments in each dimension. This array must have at least a length of two and have valid values in the first two array locations or a fatal error occurs.

minGlobalCoordPerDim Array of minimum physical coordinates in each dimension.

[maxGlobalCoordPerDim] Array of maximum physical coordinates in each direction.

[deltaPerDim] Array of constant physical increments in each direction.

[horzstagger] `ESMF_GridHorzStagger` specifier to denote horizontal grid stagger.

[dimNames] Array of dimension names.

[dimUnits] Array of dimension units.

[coordorder] `ESMF_CoordOrder` specifier to denote coordinate ordering.

[coordindex] `ESMF_CoordIndex` specifier to denote global or local indexing.

[periodic] Logical specifier (array) to denote periodicity along the coordinate axes.

[name] `ESMF_Grid` name.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

26 IOSpec Class

26.1 Description

The IOSpec in a simple class that specifies the options for an IO activity. An important choice is the IO format. Currently only netCDF is supported. Other options include whether IO should be written to a single file or multiple files, the Fortran unit number, and the filename. The IO activity can be identified as being a restart write `ESMF_IO_RESTART` or a history write `ESMF_IO_HISTORY`, if desired.

26.2 Use and Examples

The IOSpec can be used in two ways. The first way an IOSpec can be used is by passing it into the creation method of a data class such as a Field or Bundle. This sets a default IOSpec for the data object. Any IO method that involves the data object will use the settings in the default IOSpec, as long as there is no other IO specification that overrides it. This brings us to the second way to use an IOSpec. This is not implemented for all data classes throughout ESMF yet; only Fields can write out data.
The second mode of usage is to pass an IOSpec into a particular IO method, such as an `ESMF_FieldWrite()` call. The IOSpec passed into a write or read call overrides any default settings that were set up at data object creation.

26.3 Restrictions and Future Work

1. **Limited support for archival formats.** The IOSpec does not support archival formats besides binary and netCDF. We anticipate adding support for HDF variants, GRIB, and BUFR in the future.

26.4 Class API

26.4.1 ESMF_IOSpecGet - Get values in an IOSpec

INTERFACE:

```
subroutine ESMF_IOSpecGet(iospec, filename, iofileformat, &
                          iorwtype, asyncIO, rc)
```

PARAMETERS:

```
type (ESMF_IOSpec), intent(in) :: iospec
character(len=*), intent(out), optional :: filename
type (ESMF_IOFileFormat), intent(out), optional :: iofileformat
type (ESMF_IORWType), intent(out), optional :: iorwtype
logical, intent(out), optional :: asyncIO
integer, intent(out), optional :: rc
```

DESCRIPTION:

(insert documentation here.)
!REQUIREMENTS:

26.4.2 ESMF_IOSpecSet - Set values in an IOSpec

INTERFACE:

```
      subroutine ESMF_IOSpecSet(iospec, filename, iofileformat, &
                                iorwtype, asyncIO, rc)
```

PARAMETERS:

```
      type (ESMF_IOSpec), intent(inout) :: iospec
      character(len=*), intent(in), optional :: filename
      type (ESMF_IOFileFormat), intent(in), optional :: iofileformat
      type (ESMF_IORWType), intent(in), optional :: iorwtype
      logical, intent(in), optional :: asyncIO
      integer, intent(out), optional :: rc
```

DESCRIPTION:

(insert documentation here.)
!REQUIREMENTS:

27 Overview of Distributed Data Methods

Bundles, Fields, and Arrays all have versions of the following data communication methods. In ESMF, data is communicated between DEs. Depending on the underlying communication mechanism, this may translate within the framework to a data copy, an MPI call, or something else.
There is a common object handle, an `ESMF_RouteHandle`, which allows communication patterns to be precomputed during initialization and the information stored in that RouteHandle. By specifying a RouteHandle at execution time, only the source and destination data pointers must be supplied and the runtime overhead is minimized.

27.1 Higher Level Functions

The following three methods are intended to map closely to needs of applications programs. They represent higher level communications and are described in more detail in the following sections. They are:

- **Halo** Halo operations update ghost-cell or halo regions at the boundaries of a local data decomposition.
- **Regrid** Regrid methods transform data from one Grid to another.
- **Redist** Redistribution methods move data associated with a single Grid but with different decompositions.

27.2 Lower Level Functions

The following methods correspond closely to the lower level MPI communications primitives. They are:

- **Gather** Reassembling data which is decomposed over a set of DEs into a single block of data on one DE.
- **AllGather** Reassembling data which is decomposed over a set of DEs into multiple copies of a single block of data, one copy per original DE.
- **Scatter** Spreading an undecomposed block of data on one DE over a set of DEs, decomposing that single block into smaller subsets of data, one data decomposition per DE.
- **AlltoAll** Spreading an undecomposed block of data from multiple DEs onto each of the other DEs in the set, resulting in a set of multiple decomposed data blocks per DE, one from each of the original source DEs.
- **Broadcast** Spreading an undecomposed block of data from one DE onto all other DEs, where the resulting data is still undecomposed and simply copied to all other DEs.
- **Reduction** Computing a single data value, e.g. the data maximum, minimum, sum, etc from a group of decomposed data blocks across a set of DEs, where the result is delivered to a single DE.
- **AllReduce** Computing a single data value, e.g. the data maximum, minimum, sum, etc from a group of decomposed data blocks across a set of DEs, where the result is delivered to all DEs in the set.

28 Halo Method

28.1 Description

Halo operations update ghost cell or halo regions at the boundaries of a local data decomposition. Halo regions are to be considered read-only by the local process; their data values can be used to compute the new values for cells which are local to this process, but they cannot be updated except by a halo operation. Haloing is supported at the Array and Field level. The description of halo regions that follows is phrased in terms of Arrays, but also holds for Fields (which contain Arrays).

28.2 Halo Domains

Array objects can have an optional **halo width** which defines what part of the Array is the **exclusive domain**, the **computational domain**, and the **total domain**. With no halo region, all these are the same and equal to the total size of the Array. The domains are defined as follows.

- **Exclusive** The exclusive domain is the subset of the Array which is never read by any other DE.
- **Computational** The computational domain is the subset of the Array which is read and written by the current DE.
- **Total** The total domain includes the region where data is updated from another DE during a halo operation and read but not updated by the current DE.

Figure 4 illustrates these concepts.
Halo domain information must be stored at the Array level to support operations such as the gather, which collects decomposed parts of a logically contiguous object onto a single DE. Only the computational domain is copied since the halo regions are duplicated data. The exclusive domain is guaranteed to never be the source of data for a halo operation, so no synchronization of updates to those data items needs to be done. The total domain is the actual memory size allocated for the Array, and is used when computing offsets for subdomains within the Array.

29 Regrid Method

29.1 Description

Bundle, Field, and Array classes all have regrid methods that transform their data from one `ESMF_Grid` to another. Regrid operations compute addresses and interpolation weights for remapping between different grids. All the information necessary to perform a regridding, including `ESMF_Routes` to collect non-local data and the addresses and weights, are contained in the `ESMF_RouteHandle` which is returned to the user. Since interpolation weights are based solely on the grids' geometries and addresses are stored as offsets, regrids can be shared by data classes providing they have the same `ESMF_Rellocs`. The algorithms and some of the implementation in ESMF's regridding routines are adapted from a software package called SCRIP that was developed at the Los Alamos National Laboratory by Phil Jones. However, SCRIP is a serial code and the ESMF regridding routines have been parallelized.

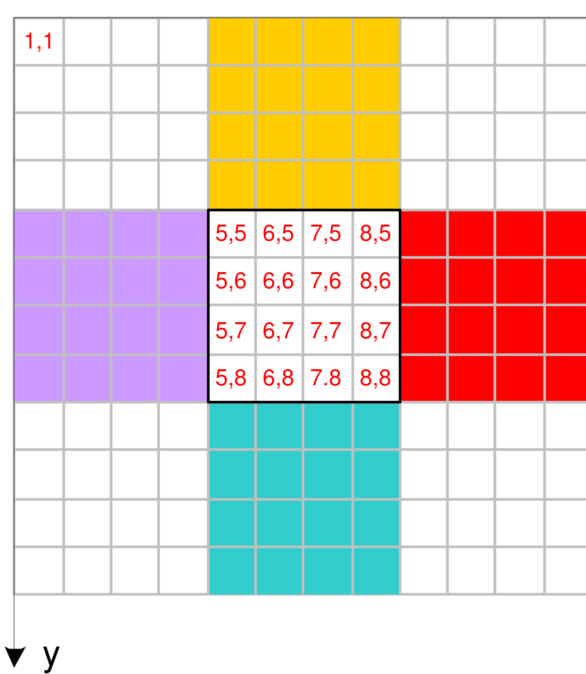
29.2 Regrid Options

29.2.1 ESMF_RegridMethod

DESCRIPTION:
General Regrid methods supported by ESMF.
Valid values are:

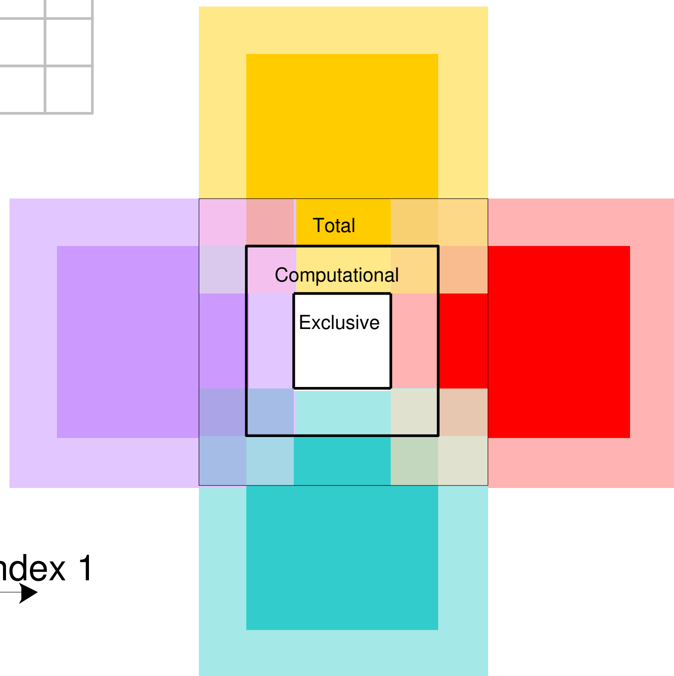
ESMF_REGRID_METHOD_BICUBIC not implemented yet

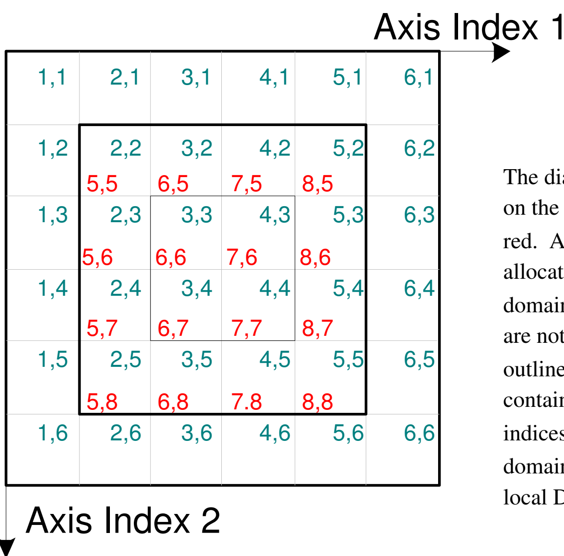
Figure 4: Diagram showing how ESMF exclusive, computational, and total domains are defined.



This example shows a grid with a length of 12 cells along the x-axis and a length of 12 cells along the y-axis. The grid is decomposed over a DELayout that has a length of 3 DEs along the x-axis and 3 DEs along the y-axis. We will look at how halo domains are defined for the data assigned to the central DE. The indices shown in red are global grid indices in the form (x,y).

We will assume that the data on each DE depends on a nearest neighbor in each direction (N,S,E,W). In order to perform computations efficiently, we would like this data on the local DE. To do this we specify a halo width of 1 cell in all directions for the data on each DE at Array or Field creation. Extra memory is allocated to hold the replicated grid cells.





The diagram directly left shows index values for the data on the central DE. The global grid indices are shown in red. Axis indices, which correspond to the memory allocated on the DE, are shown in blue. The **exclusive** domain is the inner black square; it contains grid cells that are not replicated on any other DEs. The black square outlined darkly in black is the **computational** domain; it contains all of the grid cells that have unique global grid indices and are updated by the local DE. The **total** domain is the entire extent of the memory allocated on the local DE.

Like the bilinear remapping, bicubic remapping is applicable only for logically-rectangular or block-structured logically-rectangular grids. The bicubic remapping exactly follows the bilinear remapping except that four weights for each corner point are required. Thus, num_wts is set to four for this option. The bicubic remapping is

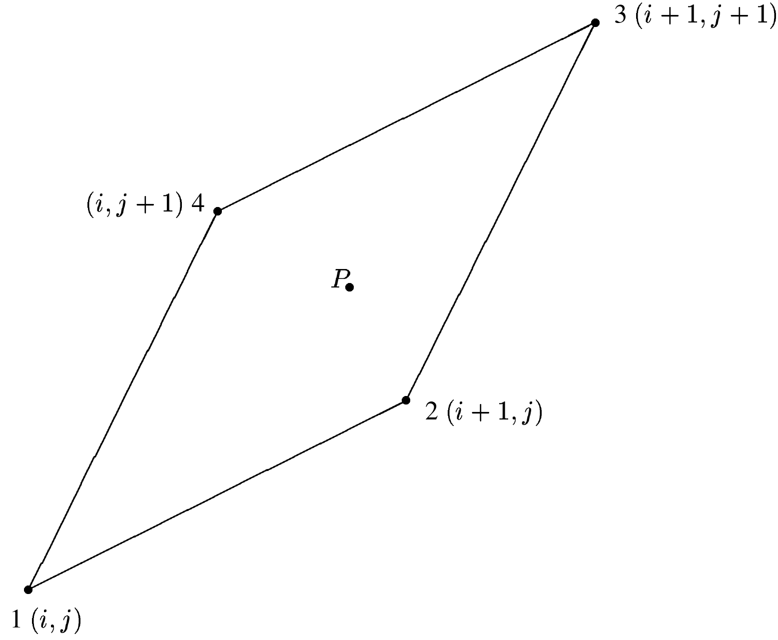
$$
\begin{aligned}
f_P \;=\;& (1-\beta^2(3-2\beta))(1-\alpha^2(3-2\alpha))f(i,j) + \\
& (1-\beta^2(3-2\beta))\alpha^2(3-2\alpha)f(i+1,j) + \\
& \beta^2(3-2\beta)\alpha^2(3-2\alpha)f(i+1,j+1) + \\
& \beta^2(3-2\beta)(1-\alpha^2(3-2\alpha))f(i,j+1) + \\
& (1-\beta^2(3-2\beta))\alpha(\alpha-1)^2\frac{\partial f}{\partial i}(i,j) + \\
& (1-\beta^2(3-2\beta))\alpha^2(\alpha-1)\frac{\partial f}{\partial i}(i+1,j) + \\
& \beta^2(3-2\beta)\alpha^2(\alpha-1)\frac{\partial f}{\partial i}(i+1,j+1) + \\
& \beta^2(3-2\beta)\alpha(\alpha-1)^2\frac{\partial f}{\partial i}(i,j+1) + \\
& \beta(\beta-1)^2(1-\alpha^2(3-2\alpha))\frac{\partial f}{\partial j}(i,j) + \\
& \beta(\beta-1)^2\alpha^2(3-2\alpha)\frac{\partial f}{\partial j}(i+1,j) + \\
& \beta^2(\beta-1)\alpha^2(3-2\alpha)\frac{\partial f}{\partial j}(i+1,j+1) + \\
& \beta^2(\beta-1)(1-\alpha^2(3-2\alpha))\frac{\partial f}{\partial j}(i,j+1) + \\
& \alpha(\alpha-1)^2\beta(\beta-1)^2\frac{\partial^2 f}{\partial i\partial j}(i,j) + \\
& \alpha^2(\alpha-1)\beta(\beta-1)^2\frac{\partial^2 f}{\partial i\partial j}(i+1,j) + \\
& \alpha^2(\alpha-1)\beta^2(\beta-1)\frac{\partial^2 f}{\partial i\partial j}(i+1,j+1) + \\
& \alpha(\alpha-1)^2\beta^2(\beta-1)\frac{\partial^2 f}{\partial i\partial j}(i,j+1)
\end{aligned} \tag{1}
$$

where α and β are identical to those found in the bilinear case and are found using an identical algorithm. Note that unlike the conservative remappings, the gradients here are gradients with respect to the *logical* variable and not latitude or longitude. Lastly, the four weights corresponding to each address pair correspond to the weight multiplying the field value at the point, the weight multiplying the gradient with respect to i, the weight multiplying the gradient with respect to j, and the weight multiplying the cross gradient in that order.

ESMF_REGRID_METHOD_BILINEAR The bilinear regridding methods uses a local bilinear approximation to interpolate to a point in a quadrilateral grid. This is applicable only for logically-rectangular or block-structured logically-rectangular grids. Standard bilinear interpolation schemes can be found in many textbooks. Here we present a more general scheme which uses a local bilinear approximation to interpolate to a point in a quadrilateral grid. Consider the grid points shown in Fig. 5 labelled with logically-rectangular indices (e.g. (i,j)).

Let the latitude-longitude coordinates of point 1 be $(\theta(i,j),\phi(i,j))$, the coordinates of point 2 be $(\theta(i+1,j),\phi(i+1,j))$, etc. Now let α and β be continuous local coordinates such that the coordinates (α,β) of

Figure 5: A general quadrilateral grid.



point 1 are $(0,0)$, point 2 are $(1,0)$, point 3 are $(1,1)$ and point 4 are $(0,1)$. If point P lies inside the cell formed by the four points above, the function f at point P can be approximated by

$$\begin{aligned} f_P &= (1-\alpha)(1-\beta)f(i,j) + \alpha(1-\beta)f(i+1,j) + \\ &\quad \alpha\beta f(i+1,j+1) + (1-\alpha)\beta f(i,j+1) \\ &= w_1 f(i,j) + w_2 f(i+1,j) + w_3 f(i+1,j+1) + w_4 f(i,j+1). \end{aligned} \tag{2}$$

The remapping weights must therefore be computed by finding α and β at point P.

The latitude-longitude coordinates (θ, ϕ) of point P are known and can also be approximated by

$$\begin{aligned} \theta &= (1-\alpha)(1-\beta)\theta_1 + \alpha(1-\beta)\theta_2 + \alpha\beta\theta_3 + (1-\alpha)\beta\theta_4 \\ \phi &= (1-\alpha)(1-\beta)\phi_1 + \alpha(1-\beta)\phi_2 + \alpha\beta\phi_3 + (1-\alpha)\beta\phi_4. \end{aligned} \tag{3}$$

Because (3) is nonlinear in α and β, we must linearize and iterate toward a solution. Differentiating (3) results in

$$\begin{bmatrix} \delta\theta \\ \delta\phi \end{bmatrix} = A \begin{bmatrix} \delta\alpha \\ \delta\beta \end{bmatrix}, \tag{4}$$

where

$$A = \begin{bmatrix} (\theta_2-\theta_1) + (\theta_1-\theta_4+\theta_3-\theta_2)\beta & (\theta_4-\theta_1) + (\theta_1-\theta_4+\theta_3-\theta_2)\alpha \\ (\phi_2-\phi_1) + (\phi_1-\phi_4+\phi_3-\phi_2)\beta & (\phi_4-\phi_1) + (\phi_1-\phi_4+\phi_3-\phi_2)\alpha \end{bmatrix}. \tag{5}$$

Inverting this system,

$$\delta\alpha = \begin{vmatrix} \delta\theta & (\theta_4 - \theta_1) + (\theta_1 - \theta_4 + \theta_3 - \theta_2)\alpha \\ \delta\phi & (\phi_4 - \phi_1) + (\phi_1 - \phi_4 + \phi_3 - \phi_2)\alpha \end{vmatrix} \div \det(A), \tag{6}$$

and

$$\delta\beta = \begin{vmatrix} (\theta_2 - \theta_1) + (\theta_1 - \theta_4 + \theta_3 - \theta_2)\beta & \delta\theta \\ (\phi_2 - \phi_1) + (\phi_1 - \phi_4 + \phi_3 - \phi_2)\beta & \delta\phi \end{vmatrix} \div \det(A). \tag{7}$$

Starting with an initial guess for α and β (say $\alpha = \beta = 0$), equations (6) and (7) can be iterated until $\delta\alpha$ and $\delta\beta$ are suitably small. The weights can then be computed from (2). Note that for simple latitude-longitude grids, this iteration will converge in the first iteration.

In order to compute the weights using this general bilinear iteration, it must be determined in which box the point P resides. For this, the search algorithms outlined in the previous chapter are used with the exception that instead of using cell corners, the relevant box is formed by neighbor cell centers as shown in Fig. 5.

ESMF_REGRID_METHOD_CONSERV1 First-order and second-order conservative remapping share a common algorithm, though currently only first-order has been implemented. ESMF implements a conservative remapping scheme described in detail elsewhere [2]. A brief outline will be given here to aid the user in understanding this regridding algorithm.

To compute a flux on a new (destination) grid which results in the same energy or water exchange as a flux f on an old (source) grid, the destination flux F at a destination grid cell k must satisfy

$$\overline{F}_k = \frac{1}{A_k} \int \int_{A_k} f dA, \tag{8}$$

where $\overline{F}$ is the area-averaged flux and A_k is the area of cell k. Because the integral in (8) is over the area of the destination grid cell, only those cells on the source grid that are covered at least partly by the destination grid cell contribute to the value of the flux on the destination grid. If cell k overlaps N cells on the source grid, the remapping can be written as

$$\overline{F}_k = \frac{1}{A_k} \sum_{n=1}^{N} \int \int_{A_{nk}} f_n dA, \tag{9}$$

where A_{nk} is the area of the source grid cell n covered by the destination grid cell k, and f_n is the local value of the flux in the source grid cell (see Figure 6). Note that (9) is normalized by the destination area A_k corresponding to the `ESMF_RegridNormOpt` value of `ESMF_REGRID_NORM_DSTAREA`. The sum of the weights for a destination cell k in this case would be between 0 and 1 and would be the area fraction if f_n were identically 1 everywhere on the source grid. The normalization option `ESMF_REGRID_NORM_FRACAREA` would actually divide by the area of the source grid overlapped by cell k:

$$\sum_{n=1}^{N} \int \int_{A_{nk}} dA. \tag{10}$$

For this normalization option, remapping a function f which is 1 everywhere on the source grid would result in a function F that is exactly one wherever the destination grid overlaps a non-masked source grid cell and zero otherwise. A normalization option of `ESMF_REGRID_NORM_NONE` would result in the actual angular area participating in the remapping.

Assuming f_n is constant across a source grid cell, (9) would lead to the first-order area-weighted schemes used in current coupled models. A more accurate form of the remapping is obtained by using

$$f_n = \overline{f}_n + \nabla_n f \cdot (\vec{r} - \vec{r}_n), \tag{11}$$

where $\nabla_n f$ is the gradient of the flux in cell n and $\vec{r}_n$ is the centroid of cell n defined by

$$\vec{r}_n = \frac{1}{A_n} \int\int_{A_n} \vec{r} dA. \tag{12}$$

Such a distribution satisfies the conservation constraint and is equivalent to the first terms of a Taylor series expansion of f around $\vec{r}_n$. The remapping is thus second-order accurate if $\nabla_n f$ is at least a first-order approximation to the gradient.

The remapping can now be expanded in spherical coordinates as

$$\overline{F}_k = \sum_{n=1}^{N} \left[\overline{f}_n w_{1nk} + \left(\frac{\partial f}{\partial \theta} \right)_n w_{2nk} + \left(\frac{1}{\cos\theta} \frac{\partial f}{\partial \phi} \right)_n w_{3nk} \right], \tag{13}$$

where θ is latitude, ϕ is longitude and the three remapping weights are

$$w_{1nk} = \frac{1}{A_k} \int\int_{A_{nk}} dA, \tag{14}$$

$$\begin{aligned} w_{2nk} &= \frac{1}{A_k} \int\int_{A_{nk}} (\theta - \theta_n) dA \\ &= \frac{1}{A_k} \int\int_{A_{nk}} \theta dA - \frac{w_{1nk}}{A_n} \int\int_{A_n} \theta dA, \end{aligned} \tag{15}$$

and

$$\begin{aligned} w_{3nk} &= \frac{1}{A_k} \int\int_{A_{nk}} \cos\theta (\phi - \phi_n) dA \\ &= \frac{1}{A_k} \int\int_{A_{nk}} \phi \cos\theta dA - \frac{w_{1nk}}{A_n} \int\int_{A_n} \phi \cos\theta dA. \end{aligned} \tag{16}$$

Again, if the gradient is zero, (13) reduces to a first-order area-weighted remapping.

The area integrals in equations (14)–(16) are computed by converting the area integrals into line integrals using the divergence theorem. Computing line integrals around the overlap regions is much simpler; one simply integrates first around every grid cell on the source grid, keeping track of intersections with destination grid lines, and then one integrates around every grid cell on the destination grid in a similar manner. After the sweep of each grid, all overlap regions have been integrated.

Choosing appropriate functions for the divergence, the integrals in equations (14)–(16) become

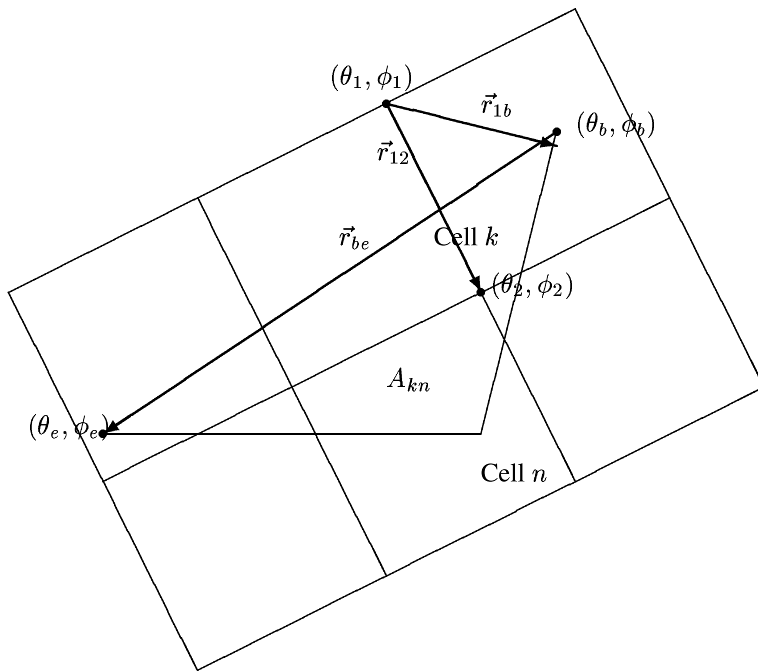
$$\int\int_{A_{nk}} dA = \oint_{C_{nk}} -\sin\theta d\phi, \tag{17}$$

$$\int\int_{A_{nk}} \theta dA = \oint_{C_{nk}} [-\cos\theta - \theta\sin\theta] d\phi, \tag{18}$$

$$\int\int_{A_{nk}} \phi\cos\theta dA = \oint_{C_{nk}} -\frac{\phi}{2} [\sin\theta\cos\theta + \theta] d\phi, \tag{19}$$

where C_{nk} is the counterclockwise path around the region A_{nk}. Computing these three line integrals during the sweeps of each grid provides all the information necessary for computing the remapping weights.

Figure 6: An example of a triangular destination grid cell k overlapping a quadrilateral source grid. The region A_{kn} is where cell k overlaps the quadrilateral cell n. Vectors used by search and intersection routines are also labelled.



As described above, the algorithm for computing the remapping weights is relatively simple. The process amounts to finding the location of the endpoint of a segment and then finding the next intersection with the other grid. The line integrals are then computed and summed according to which grid cells are associated with that particular subsegment. The most time-consuming portion of the algorithm is finding which cell on one grid contains an endpoint from the other grid. Currently in ESMF, we restrict the search to cells on the source grid who fall within the bounding box of the local piece of the decomposed destination grid. More optimal search algorithms may be added later.

Once the search has been restricted, a robust algorithm that works for most cases is a cross-product test. In this test, a cross product is computed between the vector corresponding to a cell side ($\vec{r}_{12}$ in Figure 6) and a vector extending from the beginning of the cell side to the search point ($\vec{r}_{1b}$). If

$$\vec{r}_{12} \times \vec{r}_{1b} > 0, \tag{20}$$

the point lies to the left of the cell side. If (20) holds for every cell side, the point is enclosed by the cell. This test is not completely robust and will fail for grid cells that are non-convex.

Once the location of an initial endpoint is found, it is necessary to check to see if the segment intersects with the cell side. If the segment is parametrized as

$$\begin{aligned} \theta &= \theta_b + s_1(\theta_e - \theta_b) \\ \phi &= \phi_b + s_1(\phi_e - \phi_b) \end{aligned} \tag{21}$$

and the cell side as

$$\begin{aligned} \theta &= \theta_1 + s_2(\theta_2 - \theta_1) \\ \phi &= \phi_1 + s_2(\phi_2 - \phi_1), \end{aligned} \tag{22}$$

where $\theta_1, \phi_1, \theta_2, \phi_2, \theta_b$, and θ_e are endpoints as shown in Figure 6, the intersection of the two lines occurs when θ and ϕ are equal. The linear system

$$\begin{bmatrix} (\theta_e - \theta_b) & (\theta_1 - \theta_2) \\ (\phi_e - \phi_b) & (\phi_1 - \phi_2) \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} (\theta_1 - \theta_b) \\ (\phi_1 - \phi_b) \end{bmatrix} \tag{23}$$

is then solved to determine s_1 and s_2 at the intersection point. If s_1 and s_2 are between zero and one, an intersection occurs with that cell side.

It is important also to compute identical intersections during the sweeps of each grid. To ensure that this will occur, the entire line segment is used to compute intersections rather than using a previous or next intersection as an endpoint.

Often, pairs of grids will share common lines (e.g. the Equator). When this is the case, the method described above will double-count the contribution of these line segments. Coincidences can be detected when computing cross products for the search algorithm described above. If the cross product is zero in this case, the endpoint lies on the cell side. A second cross product between the line segment and the cell side can then be computed. If the second cross product is also zero, the lines are coincident. Once a coincidence has been detected, the contribution of the coincident segment can be computed during the first sweep and ignored during the second sweep.

Some aspects of the spherical coordinate system introduce additional problems for the method described above. Longitude is multiple valued on one line on the sphere, and this branch cut may be chosen differently by different grids. Care must be taken when calculating intersections and line integrals to ensure that the proper longitude values are used. A simple method is to always check to make sure the longitude is in the same interval as the source grid cell center.

Another problem with computing weights in spherical coordinates is the treatment of the pole. First, note that although the pole is physically a point, it is a line in latitude-longitude space and has a nonzero contribution to the weight integrals. If a grid does not contain the pole explicitly as a grid vertex, the pole contribution must be added to the appropriate cells. The pole contribution can be computed analytically.

The pole also creates problems for the search and intersection algorithms described above. For example, a grid cell that overlaps the pole can result in a nonconvex cell in latitude-longitude coordinates. The cross-product test described above will fail in this case. In addition, segments near the pole typically exhibit large changes in longitude even for very short segments. In such a case, the linear parametrizations used above result in inaccuracies for determining the correct intersections.

To avoid these problems, a coordinate transformation can be used poleward of a given threshold latitude (typically within one degree of the pole). A possible transformation is the Lambert equivalent azimuthal projection

$$\begin{aligned} X &= 2\sin\left(\frac{\pi}{4}-\frac{\theta}{2}\right)\cos\phi \\ Y &= 2\sin\left(\frac{\pi}{4}-\frac{\theta}{2}\right)\sin\phi \end{aligned} \tag{24}$$

for the North Pole. The transformation for the South Pole is similar. This transformation is only used to compute intersections; line integrals are still computed in latitude-longitude coordinates. Because intersections computed in the transformed coordinates can be different from those computed in latitude-longitude coordinates, line segments which cross the latitude threshold must be treated carefully. To compute the intersections consistently for such a segment, intersections with the threshold latitude are detected and used as a normal grid intersection to provide a clean break between the two coordinate systems.

ESMF_REGRID_METHOD_LINEAR This is a standard linear regridding algorithm for 1-d grids only. In ESMF, it is used to regrid between vertical grids.

ESMF_REGRID_METHOD_NONE No regridding or undefined regrid.

29.2.2 ESMF_RegridNormOpt

DESCRIPTION:
Regrid normalization options supported by ESMF, for conservative regridding only.
Valid values are:

ESMF_REGRID_NORM_DSTAREA The Regrid weights are normalized by the destination area of each cell.

ESMF_REGRID_NORM_FRACAREA The Regrid weights are normalized by the area of the source grid overlapped by each cell (default).

ESMF_REGRID_NORM_NONE No normalization applied to Regrid weights.

ESMF_REGRID_NORM_UNKNOWN Unknown or undefined normalization.

29.3 Design and Implementation Notes

Regrid has been designed to be as efficient as possible during its Run routine. Although the initial calculation during the Store routines can be computationally intensive, the `ESMF_RouteHandle` object it creates is designed to be reused by similar Fields on the same Grids. And, as long as the Grids are static, RegridStore can be called once and reused throughout a simulation. It leverages internal structures and methods used throughout ESMF for communication so that algorithmic and programming improvements can be focused on a single location.
Because many methods are supported for regridding, the main Store function branches to a specific creation function based on the regrid method requested (e.g. bilinear, conservative, spectral). Each of these regrid methods are in a separate module to prevent the main Regrid module from becoming too large. The user is unaware of this hierarchy as the top-level module provides a unified API.
The RouteHandle object created by the RegridStore function contains a set of "links" which identify how a Field at a point on the destination Grid is related to a Field at a point on the source Grid. As such, a "link" consists of a source address, a destination address and a weight. The addresses are stored as indices to allow reuse by different Fields on the same Grids. Because the Grids are generally distributed very differently, the Regrid object also contains communication information for any data motion required for the regridding.

29.3.1 Parallel Implementation

On parallel processing platforms, only a piece of each Grid is represented locally on a particular DE. In order to calculate a regrid, the portions of the decomposed source Grid that overlaps the local destination Grid must be gathered to its DE. During the RegridStore routine, ESMF determines the list of the source DE's and Grid extents that must be collected to cover the destination Grid and stores them in an `ESMF_DomainList`. Currently ESMF gathers all the data from each DE whose source Grid intersects the local destination Grid, but there has been some work in identifying and collecting subsets of distributed data instead. The communication pattern to gather the corresponding data is stored in an `ESMF_Route` for use during the run routine. Each DE then stores the Route and list of links unique to its destination Grid.
During the run routine, the source data is first gathered using the precomputed Route and stored as a 1-D vector. The main calculation is then a loop over the number of links that effectively sums the product of the source data and the interpolation weights and loads the result into the corresponding destination Array address.

29.3.2 Regrid Objects

The `ESMF_Regrid` object itself is relatively small, containing mostly just pointers to the source and destination Field information and settings for regrid options. Regrid methods also create an `ESMF_TransformValues` object, which holds information about the "links", including the number of them and Arrays of source addresses, destination addresses, and interpolation weights. Both the Regrid and TransformValues objects are private and contained by the RouteHandle object.

29.3.3 Restrictions and Future Work

1. **Support is limited to 1D and 2D regridding.** Regridding support is limited to two dimensions.
2. **Masks are not implemented.** Regridding methods take masks in their argument lists, but they have no effect.
3. **Future regrid methods.** The following methods will be added:

   ESMF_REGRID_METHOD_ADJOINT Create adjoint of existing regrid

   ESMF_REGRID_METHOD_FILE Read a regrid from a file

   ESMF_REGRID_METHOD_FOURIER Fourier transform

   ESMF_REGRID_METHOD_INDEX Index-space regrid (shift, stencil)

   ESMF_REGRID_METHOD_LEGENDRE Legendre transform

   ESMF_REGRID_METHOD_NEARNBR Nearest-neighbor dist-weighted avg

   ESMF_REGRID_METHOD_RASTER Regrid by rasterizing domain

   ESMF_REGRID_METHOD_REGRIDCOPY Copy existing regrid

   ESMF_REGRID_METHOD_SHIFT Shift addresses of existing regrid

   ESMF_REGRID_METHOD_SPLINE Cubic spline for 1-d regridding

   ESMF_REGRID_METHOD_USER User-supplied method

```
! !PROGRAM: ESMF_RegridEx - Using the Regridding methods
!
! !DESCRIPTION:
!
! This program shows examples of using Regrid on Field data
!-----------------------------------------------------------------------------

    ! ESMF Framework module
    use ESMF_Mod

    implicit none
```

```
! Local variables
type(ESMF_Field) :: field1, field2
type(ESMF_Grid) :: srcgrid, dstgrid
type(ESMF_RouteHandle) :: regrid_rh
type(ESMF_Array) :: arraya, arrayb
type(ESMF_DELayout) :: layout1, layout2
integer :: rc
```

29.3.4 Precomputing and Executing a Regrid

The user has already created an `ESMF_Grid`, an `ESMF_Array` with data, and put them together in an `ESMF_Field`. An `ESMF_RouteHandle` is created and the data movement needed to execute the regrid is stored with that handle by the store method. To actually execute the operation, the source and destination data objects must be supplied, along with the same `ESMF_RouteHandle`.

```
regrid_rh = ESMF_RouteHandleCreate(rc)


call ESMF_FieldRegridStore(field1, field2, layout1, &
                           routehandle=regrid_rh, &
                           regridmethod=ESMF_REGRID_METHOD_BILINEAR, rc=rc)


call ESMF_FieldRegrid(field1, field2, regrid_rh, rc=rc)


call ESMF_FieldRegridRelease(regrid_rh, rc=rc)


call ESMF_RouteHandleDestroy(regrid_rh)
```

30 Redist Method

30.1 Description

Redistribution operations always operate on data associated with a single Grid, but are either decomposed across multiple DEs differently, or have different index orderings, or are decomposed in different dimensions. The Redistribution methods involve only data movement; no interpolation, data binning, or averaging is performed. A typical example of redistribution is the data transpose associated with the translation between physical and Fourier space.
An example of the use of ESMF redistribution methods is presented in the FieldRedist system test.

Part IV
Infrastructure: Utilities

31 Overview of Infrastructure Utility Classes

The ESMF utilities are a set of tools for quickly assembling modeling applications. The Time Management Library provides utilities for time and date representation and calculation, and higher-level utilities that control model time stepping and alarming.
The Array class offers an efficient, language-neutral way of storing and manipulating data arrays.
The Communications/Memory/Kernel library provides utilities for isolating system-dependent functions to ease platform portability. It provides services to represent a particular machine's characteristics and to organize these into processor lists and layouts to allow for optimal allocation of resources to an ESMF component. Also provided is a unified interface for system-dependent communication services such as MPI or pthreads.
ESMF Configuration Management is based on NASA DAO's Inpak package, a collection of routines for accessing files containing input parameters stored in an ASCII format.

32 Time Manager Utility

The ESMF Time Manager utility includes software for time and date representation and calculations, model time advancement, and the identification of unique and periodic events. Since multi-component geophysical applications often require synchronization across the time management schemes of the individual components, the Time Manager's standard calendars and consistent time representation promote component interoperability.

Key Features
Drift-free timekeeping through an integer-based internal time representation.
The ability to represent time as a rational fraction, to support exact timekeeping in applications that involve grid refinement.
Support for many calendar types, including user-customized calendars.
Support for both concurrent and sequential modes of component execution.
Support for varying and negative time steps.

32.1 Time Manager Classes

There are five ESMF classes that represent time concepts:

- **Calendar** A Calendar can be used to keep track of the date as an ESMF Gridded Component advances in time. Standard calendars (such as Gregorian and 360-day) and user-specified calendars are supported. Calendars can be queried for quantities such as seconds per day, days per month, and days per year.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:31pm EST in the Gregorian calendar. The Time class can be used to represent the start and stop time of a time integration.
- **TimeInterval** TimeIntervals represent a period of time, such as 300 milliseconds. Time steps can be represented using TimeIntervals.
- **Clock** Clocks collect the parameters and methods used for model time advancement into a convenient package. A Clock can be queried for quantities such as start time, stop time, current time, and time step. Clock methods include incrementing the current time, and determining if it is time to stop.
- **Alarm** Alarms identify unique or periodic events by "ringing" - returning a true value - at specified times. For example, an Alarm might be set to ring on the day of the year when leaves start falling from the trees in a climate model.

The ESMF Time Manager utility includes software to manage model calendars, advance model time, and perform time and date calculations. The software classes that handle these functions are **Times**, **TimeIntervals**, **Clocks**, **Alarms**, and **Calendars**.

In the remainder of this section, we briefly summarize the functionality that the Time Manager classes provide. Detailed descriptions and usage examples precede the API listing for each class.

32.2 Calendar

An ESMF Calendar can be queried for seconds per day, days per month and days per year. The flexible definition of Calendars allows them to be defined for planetary bodies other than Earth. The set of supported calendars includes:

Gregorian The standard Gregorian calendar.

no-leap The Gregorian calendar with no leap years.

Julian Day A Julian days calendar.

360-day A 30-day-per-month, 12-month-per-year calendar.

no calendar Tracks only elapsed model time in seconds.

See Section 33.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

32.3 Time Instants and Time Intervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. TimeIntervals support operations such as add, subtract, compare size, reset value, copy value, and subdivide by a scalar. Times, which are moments in time associated with specific Calendars, can be incremented or decremented by TimeIntervals, compared to determine which of two Times is later, differenced to obtain the TimeInterval between two Times, copied, reset, and manipulated in other useful ways. Times support a host of different queries, both for values of individual Time components such as year, month, day, and second, and for derived values such as day of year, middle of current month and Julian day. It is also possible to retrieve the value of the hardware realtime clock in the form of a Time. See Sections 34.1 and 35.1, respectively, for use and examples of Times and TimeIntervals.
Since climate modeling, numerical weather prediction and other Earth and space applications have widely varying time scales and require different sorts of calendars, Times and TimeIntervals must support a wide range of time specifiers, spanning nanoseconds to years. The interfaces to these time classes are defined so that the user can specify a time using a combination of units selected from the list shown in Table 1.

32.4 Clocks and Alarms

Although it is possible to repeatedly step a Time forward by a TimeInterval using arithmetic on these basic types, it is useful to identify a higher-level concept to represent this function. We refer to this capability as a Clock, and include in its required features the ability to store the start and stop times of a model run, to check when time advancement should cease, and to query the value of quantities such as the current time and the time at the previous time step. The Time Manager includes a class with methods that return a true value when a periodic or unique event has taken place; we refer to these as Alarms. Applications may contain temporary or multiple Clocks and Alarms. Sections 36.1 and 37.1 describe the use of Clocks and Alarms in detail.

Table 1: Specifiers for Times and TimeIntervals

<table>
<tr><th>Unit</th><th>Meaning</th></tr>
<tr><td><yy|yy_i8></td><td>Year.</td></tr>
<tr><td>mm</td><td>Month of the year.</td></tr>
<tr><td>dd</td><td>Day of the month.</td></tr>
<tr><td><d|d_i8|d_r8></td><td>Julian day.</td></tr>
<tr><td><h|h_r8></td><td>Hour.</td></tr>
<tr><td><m|m_r8></td><td>Minute.</td></tr>
<tr><td><s|s_i8|s_r8></td><td>Second.</td></tr>
<tr><td><ms|ms_r8></td><td>Millisecond.</td></tr>
<tr><td><us|us_r8></td><td>Microsecond.</td></tr>
<tr><td><ns|ns_r8></td><td>Nanosecond.</td></tr>
<tr><td>O</td><td>Time zone offset in integer number of hours and minutes.</td></tr>
<tr><td>sN</td><td>Numerator for times of the form s $+\frac{sN}{sD}$, where s is seconds and s, sN, and sD are integers. This format provides a mechanism for supporting exact behavior.</td></tr>
<tr><td>sD</td><td>Denominator for times of the form s $+\frac{sN}{sD}$, where s is seconds and s, sN, and sD are integers.</td></tr>
</table>

32.5 Design and Implementation Notes

1. **Base TimeIntervals and Times on the same integer representation.** It is useful to allow both TimeIntervals and Times to inherit from a single class, BaseTime. In C++, this can be implemented by using inheritance. In Fortran, it can be implemented by having the derived types TimeIntervals and Times contain a derived type BaseTime. In both cases, the BaseTime class can be made private and invisible to the user.

   The result of this strategy is that Time Intervals and Times gain a consistent core representation of time as well a set of basic methods.

   The BaseTime class can be designed with a minimum number of elements to represent any required time. The design is based on the idea used in the real-time POSIX 1003.1b-1993 standard. That is, to represent time simply as a pair of integers: one for seconds (whole) and one for nanoseconds (fractional). These can then be converted at the interface level to any desired format.

   For ESMF, this idea can be modified and extended, in order to handle the requirements for a large time range (> 200,000 years) and to exactly represent any rational fraction, not just nanoseconds. To handle the large time range, a 64-bit or greater integer is used for whole seconds. Any rational fractional second is expressed using two additional integers: a numerator and a denominator. Both the whole seconds and fractional numerator are signed to handle negative time intervals and instants. For arithmetic consistency both must carry the same sign (both positve or both negative), except, of course, for zero values. The fractional seconds element (numerator) is bounded with respect to whole seconds. If the absolute value of the numerator becomes greater than or equal to the denominator, whole seconds are incremented or decremented accordingly and the numerator is reset to the remainder. Conversions are performed upon demand by interface methods within the TimeInterval and Time classes. This is done because different applications require different representations of time intervals and time instances.

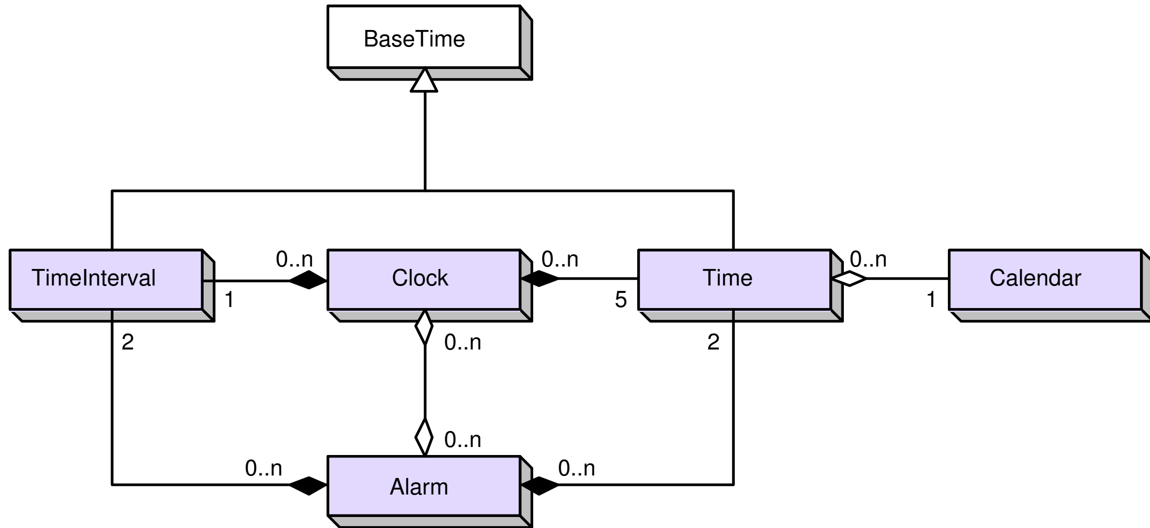
   The BaseTime class defines increment and decrement methods for basic TimeInterval calculations between Time instants. It is done here rather than in the Calendar class because it can be done with simple second-based arithmetic that is calendar independent.

   Comparison methods can also be defined in the BaseTime class. These perform equality/inequality, less than, and greater than comparisons between any two TimeIntervals or Times. These methods capture the common comparison logic between TimeIntervals and Times and hence are defined here for sharing.

2. **The Time class depends on a calendar.** The Time class contains an internal Calendar class. Upon demand by a user, the results of an increment or decrement operation are converted to user units, which may be calendar-dependent, via methods obtained from their internal Calendar.

32.6 Object Model

The following is a simplified UML diagram showing the structure of the Time Manager utility. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



33 Calendar Class

33.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, no-leap, 360-day, and no-calendar. It also supports a user-customized calendar. Brief descriptions are provided for each calendar below. For more information on standard calendars, see [4].

33.2 Calendar Options

33.2.1 ESMF_CalendarType

DESCRIPTION:
Supported calendar types.
Valid values are:

ESMF_CAL_360DAY *Valid range: machine limits*
In the 360-day calendar, there are 12 months, each of which has 30 days. Like the no-leap calendar, this is a simple approximation to the Gregorian calendar sometimes used by modelers.

ESMF_CAL_CUSTOM *Valid range: machine limits*
The user can set calendar parameters in the generic calendar.

ESMF_CAL_GREGORIAN *Valid range: 3/1/4800 BC to 10/29/292,277,019,914*
The Gregorian calendar is the calendar currently in use throughout Western countries. Named after Pope Gregory, it is a minor correction to the older Julian calendar. In the Gregorian calendar every fourth year is a leap year in which February has 29 and not 28 days; however, years divisible by 100 are not leap years unless they are also divisible by 400. As in the Julian calendar, days begin at midnight.

ESMF_CAL_JULIAN - not yet implemented *Valid range: 4713 BC onward*
The Julian calendar was introduced by Julius Caesar in 46 B.C., and reached its final form in 8 B.C. The Julian calendar differs from the Gregorian only in the determination of leap years, lacking the correction for years divisible by 100 and 400 in the Gregorian calendar. In the Julian calendar, any positive year is a leap year if divisible by 4. (Negative years are leap years if, when divided by 4, a remainder of 3 results.) Days are considered to begin at midnight.

ESMF_CAL_JULIANDAY *Valid range: -32045 to 1x10*^{14}
Julian days simply enumerate the days and fraction of a day which have elapsed since the start of the Julian era, defined as beginning at noon on Monday, 1st January of year 4713 B.C. in the Julian calendar. Julian days, unlike the dates in the Julian and Gregorian calendars, begin at noon.

ESMF_CAL_NOCALENDAR *Valid range: machine limits*
The no-calendar option simply tracks the elapsed model time in seconds.

ESMF_CAL_NOLEAP *Valid range: machine limits*
The no-leap calendar is the Gregorian calendar with no leap years - February is always assumed to have 28 days. Modelers sometimes use this calendar as a simple, close approximation to the Gregorian calendar.

33.3 Use and Examples

In most multi-component Earth system applications, the timekeeping in each component must refer to the same standard calendar in order for the components to properly synchronize. It therefore makes sense to create as few ESMF Calendars as possible, preferably one per application. A typical strategy would be to create a single Calendar at the start of an application, and use that Calendar in all subsequent calls that accept a Calendar, such as `ESMF_TimeSet`. The following example shows how to set up an ESMF Calendar.

```
! !PROGRAM: ESMF_CalendarEx - Calendar creation examples
!
```

```
! !DESCRIPTION:
!
! This program shows examples of how to create different calendar types
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate calendars
      type(ESMF_Calendar) :: gregorianCalendar
      type(ESMF_Calendar) :: julianDayCalendar

      ! local variables for Get methods
      integer(ESMF_KIND_I8) :: dl
      type(ESMF_Time) :: time

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(rc=rc)
```

33.3.1 Calendar Creation

This example shows how to create two ESMF_Calendars.

```
      ! create a Gregorian calendar
      gregorianCalendar = ESMF_CalendarCreate("Gregorian", &
                                              ESMF_CAL_GREGORIAN, rc)


      ! create a Julian Day calendar
      julianDayCalendar = ESMF_CalendarCreate("JulianDay", &
                                              ESMF_CAL_JULIANDAY, rc)
```

33.3.2 Calendar Comparison

This example shows how to compare an ESMF_Calendar with a known calendar type.

```
      ! compare calendar type against a known type
      if (gregorianCalendar == ESMF_CAL_GREGORIAN) then
          print *, "gregorianCalendar is of type ESMF_CAL_GREGORIAN."
      else
          print *, "gregorianCalendar is not of type ESMF_CAL_GREGORIAN."
      end if
```

33.3.3 Time Conversion Between Calendars

This example shows how to convert a time from one ESMF_Calendar to another.

```
      call ESMF_TimeSet(time, yy=2004, mm=4, dd=17, &
                        calendar=gregorianCalendar, rc=rc)
```

```
! switch time's calendar to perform conversion
call ESMF_TimeSet(time, calendar=julianDayCalendar, rc=rc)

call ESMF_TimeGet(time, d_i8=dl, rc=rc)
print *, "Gregorian date 2004/4/17 is ", dl, &
         " days in the Julian Day calendar."
```

33.3.4 Calendar Destruction

This example shows how to destroy two `ESMF_Calendars`.

```
call ESMF_CalendarDestroy(julianDayCalendar, rc)

call ESMF_CalendarDestroy(gregorianCalendar, rc)

! finalize ESMF framework
call ESMF_Finalize(rc)

end program ESMF_CalendarEx
```

33.4 Restrictions and Future Work

1. **Months per year set to 12.** Due to the requirement of only Earth modeling, the number of months per year is hard-coded at 12. However, for easy modification, this is implemented via a Fortran parameter and a C preprocessor #define.

33.5 Class API

33.5.1 ESMF_CalendarOperator(==) - Test if Calendar 1 is equal to Calendar 2

INTERFACE:

```
interface operator(==)
if (calendar1 == calendar2) then ... endif
             OR
result = (calendar1 == calendar2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Calendar), intent(in) :: calendar1
type(ESMF_Calendar), intent(in) :: calendar2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Calendar` class. Compare two calendar objects for equality; return true if equal, false otherwise. Comparison is based on the calendar type.
The arguments are:

calendar1 The first `ESMF_Calendar` in comparison.

calendar2 The second `ESMF_Calendar` in comparison.

33.5.2 ESMF_CalendarOperator(==) - Test if Calendar Type 1 is equal to Calendar Type 2

INTERFACE:

```
interface operator(==)
if (calendartype1 == calendartype2) then ... endif
             OR
result = (calendartype1 == calendartype2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CalendarType), intent(in) :: calendartype1
type(ESMF_CalendarType), intent(in) :: calendartype2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Calendar` class. Compare two calendar types for equality; return true if equal, false otherwise.
The arguments are:

calendartype1 The first `ESMF_CalendarType` in comparison.

calendartype2 The second `ESMF_CalendarType` in comparison.

33.5.3 ESMF_CalendarOperator(==) - Test if Calendar is equal to Calendar Type

INTERFACE:

```
interface operator(==)
if (calendar == calendartype) then ... endif
             OR
result = (calendar == calendartype)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(in) :: calendar
type(ESMF_CalendarType), intent(in) :: calendartype
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Calendar` class. Compare a calendar object's type with a given calendar type for equality; return true if equal, false otherwise.
The arguments are:

calendar The `ESMF_Calendar` in comparison.

calendartype The `ESMF_CalendarType` in comparison.

33.5.4 ESMF_CalendarOperator(==) - Test if Calendar Type is equal to Calendar

INTERFACE:

```
interface operator(==)
if (calendartype == calendar) then ... endif
             OR
result = (calendartype == calendar)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CalendarType), intent(in) :: calendartype
type(ESMF_Calendar),     intent(in) :: calendar
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Calendar` class. Compare a calendar type with a given calendar object's type for equality; return true if equal, false otherwise.
The arguments are:

calendartype The `ESMF_CalendarType` in comparison.

calendar The `ESMF_Calendar` in comparison.

33.5.5 ESMF_CalendarOperator(/=) - Test if Calendar 1 is not equal to Calendar 2

INTERFACE:

```
interface operator(/=)
if (calendar1 /= calendar2) then ... endif
             OR
result = (calendar1 /= calendar2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Calendar), intent(in) :: calendar1
type(ESMF_Calendar), intent(in) :: calendar2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Calendar` class. Compare two calendar objects for inequality; return true if not equal, false otherwise. Comparison is based on the calendar type.
The arguments are:

calendar1 The first `ESMF_Calendar` in comparison.

calendar2 The second `ESMF_Calendar` in comparison.

33.5.6 ESMF_CalendarOperator(/=) - Test if Calendar Type 1 is not equal to Calendar Type 2

INTERFACE:

```
interface operator(/=)
if (calendartype1 /= calendartype2) then ... endif
             OR
result = (calendartype1 /= calendartype2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CalendarType), intent(in) :: calendartype1
type(ESMF_CalendarType), intent(in) :: calendartype2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Calendar` class. Compare two calendar types for inequality; return true if not equal, false otherwise.
The arguments are:

calendartype1 The first `ESMF_CalendarType` in comparison.

calendartype2 The second `ESMF_CalendarType` in comparison.

33.5.7 ESMF_CalendarOperator(/=) - Test if Calendar is not equal to Calendar Type

INTERFACE:

```
interface operator(/=)
if (calendar /= calendartype) then ... endif
             OR
result = (calendar /= calendartype)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(in) :: calendar
type(ESMF_CalendarType), intent(in) :: calendartype
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Calendar` class. Compare a calendar object's type with a given calendar type for inequality; return true if equal, false otherwise.
The arguments are:

calendar The `ESMF_Calendar` in comparison.

calendartype The `ESMF_CalendarType` in comparison.

33.5.8 ESMF_CalendarOperator(/=) - Test if Calendar Type is not equal to Calendar

INTERFACE:

```
interface operator(/=)
if (calendartype /= calendar) then ... endif
             OR
result = (calendartype /= calendar)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CalendarType), intent(in) :: calendartype
type(ESMF_Calendar),     intent(in) :: calendar
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Calendar` class. Compare a calendar type with a given calendar object's type for inequality; return true if equal, false otherwise.
The arguments are:

calendartype The `ESMF_CalendarType` in comparison.

calendar The `ESMF_Calendar` in comparison.

33.5.9 ESMF_CalendarCreate - Create a new ESMF Calendar of built-in type

INTERFACE:

```
! Private name; call using ESMF_CalendarCreate()
function ESMF_CalendarCreateBuiltIn(name, calendartype, rc)
```

RETURN VALUE:

```
type(ESMF_Calendar) :: ESMF_CalendarCreateBuiltIn
```

ARGUMENTS:

```
character (len=*),       intent(in),  optional :: name
type(ESMF_CalendarType), intent(in)            :: calendartype
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Creates and sets a `calendar` to the given built-in `ESMF_CalendarType`.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarCreate()`.
The arguments are:

[name] The name for the newly created calendar. If not specified, a default unique name will be generated: "CalendarNNN" where NNN is a unique sequence number from 001 to 999.

calendartype The built-in `ESMF_CalendarType`. Valid values are: `ESMF_CAL_360DAY`, `ESMF_CAL_GREGORIAN`, `ESMF_CAL_JULIANDAY`, `ESMF_CAL_NOCALENDAR`, and `ESMF_CAL_NOLEAP`. See the "Time Manager Reference" document for a description of each calendar type.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.10 ESMF_CalendarCreate - Create a copy of an ESMF Calendar

INTERFACE:

```
! Private name; call using ESMF_CalendarCreate()
function ESMF_CalendarCreateCopy(calendar, rc)
```

RETURN VALUE:

```
type(ESMF_Calendar) :: ESMF_CalendarCreateCopy
```

ARGUMENTS:

```
type(ESMF_Calendar), intent(in)            :: calendar
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Creates a copy of a given `ESMF_Calendar`.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarCreate()`.
The arguments are:

calendar The `ESMF_Calendar` to copy.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.11 ESMF_CalendarCreate - Create a new custom ESMF Calendar

INTERFACE:

```
! Private name; call using ESMF_CalendarCreate()
function ESMF_CalendarCreateCustom(name, daysPerMonth, secondsPerDay, &
                                   daysPerYear, daysPerYearDn, &
                                   daysPerYearDd, rc)
```

RETURN VALUE:

```
type(ESMF_Calendar) :: ESMF_CalendarCreateCustom
```

ARGUMENTS:

```
character (len=*),     intent(in),  optional :: name
integer, dimension(:), intent(in),  optional :: daysPerMonth
integer(ESMF_KIND_I4), intent(in),  optional :: secondsPerDay
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYear   ! not implemented
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDn ! not implemented
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDd ! not implemented
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Creates a custom `ESMF_Calendar` and sets its properties.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarCreate()`.
The arguments are:

[name] The name for the newly created calendar. If not specified, a default unique name will be generated: "CalendarNNN" where NNN is a unique sequence number from 001 to 999.

[daysPerMonth] Integer array of days per month, for each month of the year. The number of months per year is variable and taken from the size of the array. If unspecified, months per year = 0, with the days array undefined.

[secondsPerDay] Integer number of seconds per day. Defaults to 86400 if not specified.

[daysPerYear] Integer number of days per year. Use with daysPerYearDn and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDn] Integer numerator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear (see above) and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDd] Integer denominator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear and daysPerYearDn (see above) to specify a days-per-year calendar for any planetary body. Default = 1. (Not implemented yet).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.12 ESMF_CalendarDestroy - Free resources associated with a Calendar

INTERFACE:

```
subroutine ESMF_CalendarDestroy(calendar, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar) :: calendar
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this `ESMF_Calendar`.
The arguments are:

calendar Destroy contents of this `ESMF_Calendar`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.13 ESMF_CalendarGet - Get Calendar properties

INTERFACE:

```
subroutine ESMF_CalendarGet(calendar, name, calendartype, &
                            daysPerMonth, monthsPerYear, &
                            secondsPerDay, secondsPerYear, &
                            daysPerYear, &
                            daysPerYearDn, daysPerYearDd, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(in)            :: calendar
character (len=*),       intent(out), optional :: name
type(ESMF_CalendarType), intent(out), optional :: calendartype
integer, dimension(:),   intent(out), optional :: daysPerMonth
integer,                 intent(out), optional :: monthsPerYear
integer(ESMF_KIND_I4),   intent(out), optional :: secondsPerDay
integer(ESMF_KIND_I4),   intent(out), optional :: secondsPerYear
integer(ESMF_KIND_I4),   intent(out), optional :: daysPerYear   ! not implemented
integer(ESMF_KIND_I4),   intent(out), optional :: daysPerYearDn ! not implemented
integer(ESMF_KIND_I4),   intent(out), optional :: daysPerYearDd ! not implemented
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Gets one or more of an `ESMF_Calendar`'s properties.
The arguments are:

calendar The object instance to query.

[name] The name of this calendar.

[calendartype] The `CalendarType` ESMF_CAL_GREGORIAN, ESMF_CAL_JULIANDAY, etc.

[daysPerMonth] Integer array of days per month, for each month of the year.

[monthsPerYear] Integer number of months per year; the size of the daysPerMonth array.

[secondsPerDay] Integer number of seconds per day.

[secondsPerYear] Integer number of seconds per year.

[daysPerYear] Integer number of days per year. For calendars with intercalations, daysPerYear is the number of days for years without an intercalation. For other calendars, it is the number of days in every year. (Not implemented yet).

[daysPerYearDn] Integer fractional number of days per year (numerator). For calendars with intercalations, daysPerYearDn/daysPerYea is the average fractional number of days per year (e.g. 25/100 for Julian 4-year intercalation). For other calendars, it is zero. (Not implemented yet).

[daysPerYearDd] Integer fractional number of days per year (denominator). See daysPerYearDn above. (Not implemented yet).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.14 ESMF_CalendarPrint - Print the contents of a Calendar

INTERFACE:

```
subroutine ESMF_CalendarPrint(calendar, options, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar), intent(in)            :: calendar
character (len=*),   intent(in),  optional :: options
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Prints out an `ESMF_Calendar`'s properties to `stdio`, in support of testing and debugging. The options control the type of information and level of detail.
The arguments are:

calendar `ESMF_Calendar` to be printed out.

[options] Print options. If none specified, prints all calendar property values.
"calendartype" - print the calendar's type (e.g. ESMF_CAL_GREGORIAN).
"daysPerMonth" - print the array of number of days for each month.
"daysPerYear" - print the number of days per year (integer and fractional parts).
"monthsPerYear" - print the number of months per year.
"name" - print the calendar's name.
"secondsPerDay" - print the number of seconds in a day.
"secondsPerYear" - print the number of seconds in a year.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.15 ESMF_CalendarSet - Set a Calendar to a built-in type

INTERFACE:

```
! Private name; call using ESMF_CalendarSet()
subroutine ESMF_CalendarSetBuiltIn(calendar, name, calendartype, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(inout)          :: calendar
character (len=*),       intent(in),  optional :: name
type(ESMF_CalendarType), intent(in)             :: calendartype
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets `calendar` to the given built-in `ESMF_CalendarType`.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarSet()`.
The arguments are:

calendar The object instance to initialize.

[name] The new name for this calendar.

calendartype The built-in `CalendarType`. Valid values are: `ESMF_CAL_360DAY`, `ESMF_CAL_GREGORIAN`, `ESMF_CAL_JULIANDAY`, `ESMF_CAL_NOCALENDAR`, and `ESMF_CAL_NOLEAP`. See the "Time Manager Reference" document for a description of each calendar type.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.16 ESMF_CalendarSet - Set properties of a custom Calendar

INTERFACE:

```
! Private name; call using ESMF_CalendarSet()
subroutine ESMF_CalendarSetCustom(calendar, name, daysPerMonth, &
                                  secondsPerDay, &
                                  daysPerYear, daysPerYearDn, &
                                  daysPerYearDd, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar),   intent(inout)          :: calendar
character (len=*),     intent(in),  optional :: name
integer, dimension(:), intent(in),  optional :: daysPerMonth
integer(ESMF_KIND_I4), intent(in),  optional :: secondsPerDay
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYear   ! not implemented
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDn ! not implemented
integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDd ! not implemented
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Sets properties in a custom `ESMF_Calendar`.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarSet()`.
The arguments are:

calendar The object instance to initialize.

[name] The new name for this calendar.

[daysPerMonth] Integer array of days per month, for each month of the year. The number of months per year is variable and taken from the size of the array. If unspecified, months per year = 0, with the days array undefined.

[secondsPerDay] Integer number of seconds per day. Defaults to 86400 if not specified.

[daysPerYear] Integer number of days per year. Use with daysPerYearDn and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDn] Integer numerator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear (see above) and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDd] Integer denominator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear and daysPerYearDn (see above) to specify a days-per-year calendar for any planetary body. Default = 1. (Not implemented yet).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.17 ESMF_CalendarSetDefault - Set the default Calendar type

INTERFACE:

```
! Private name; call using ESMF_CalendarSetDefault()
subroutine ESMF_CalendarSetDefaultType(calendartype, rc)
```

ARGUMENTS:

```
type(ESMF_CalendarType), intent(in)            :: calendartype
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets the default `calendar` to the given type. Subsequent Time Manager operations requiring a calendar where one isn't specified will use the internal calendar of this type.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarSetDefault()`.
The arguments are:

calendartype The calendar type to be the default.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.18 ESMF_CalendarSetDefault - Set the default Calendar

INTERFACE:

```
! Private name; call using ESMF_CalendarSetDefault()
subroutine ESMF_CalendarSetDefaultCal(calendar, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(in)            :: calendar
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets the default `calendar` to the one given. Subsequent Time Manager operations requiring a calendar where one isn't specified will use this calendar.
This is a private method; invoke via the public overloaded entry point `ESMF_CalendarSetDefault()`.
The arguments are:

calendar The object instance to be the default.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

33.5.19 ESMF_CalendarValidate - Validate a Calendar's properties

INTERFACE:

```
subroutine ESMF_CalendarValidate(calendar, options, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar), intent(in)            :: calendar
character (len=*),   intent(in),  optional :: options
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Checks whether a `calendar` is valid.
The arguments are:

calendar `ESMF_Calendar` to be validated.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

34 Time Class

34.1 Description

A Time represents a specific point in time. In order to accommodate the range of time scales in Earth system applications, Times in the ESMF an be specified in many different ways, from years to nanoseconds. The Time interface is designed so that you select one or more options from a list of time units in order to specify a Time. The options for specifying a Time are shown in Table 1.
There are Time methods defined for setting and getting a Time, incrementing and decrementing a Time by a TimeInterval, taking the difference between two Times, and comparing Times. Special quantities such as the middle of the month and the day of the year associated with a particular Time can be retrieved. There is a method for returning the Time value as a string in the ISO 8601 format YYYY-MM-DDThh:mm:ss [1].
A Time that is specified in hours, minutes, seconds, or subsecond intervals does not need to be associated with a standard calendar; a Time whose specification includes time units of a day and greater must be. The ESMF representation of a calendar, the Calendar class, is described in Section 33.1. The `ESMF_TimeSet` method is used to initialize a Time as well as associate it with a Calendar. If a Time method is invoked in which a Calendar is necessary and one has not been set, the ESMF method will return an error condition.
In the ESMF the TimeInterval class is used to represent time periods. This class is frequently used in combination with the Time class. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

34.2 Use and Examples

Times are most frequently used to represent start, stop, and current model times. The following examples show how to create, initialize, and manipulate `Time`.

```
! !PROGRAM: ESMF_TimeEx - Time initialization and manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Time initialization and manipulation
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate two times
      type(ESMF_Time) :: time1, time2

      ! instantiate a time interval
      type(ESMF_TimeInterval) :: timeinterval1

      ! local variables for Get methods
      integer :: YY, MM, DD, H, M, S

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultCalendar=ESMF_CAL_GREGORIAN, rc=rc)
```

34.2.1 Time Initialization

This example shows how to initialize an `ESMF_Time`.

```
! initialize time1 to 2/28/2000 2:24:45
call ESMF_TimeSet(time1, yy=2000, mm=2, dd=28, h=2, m=24, s=45, rc=rc)


print *, "Time1 = "
call ESMF_TimePrint(time1, "string", rc)
```

34.2.2 Time Increment

This example shows how to increment an `ESMF_Time` by an `ESMF_TimeInterval`.

```
! initialize a time interval to 2 days, 8 hours, 36 minutes, 15 seconds
call ESMF_TimeIntervalSet(timeinterval1, d=2, h=8, m=36, s=15, rc=rc)


print *, "Timeinterval1 = "
call ESMF_TimeIntervalPrint(timeinterval1, "string", rc)


! increment time1 with timeinterval1
time2 = time1 + timeinterval1

call ESMF_TimeGet(time2, yy=YY, mm=MM, dd=DD, h=H, m=M, s=S, rc=rc)
print *, "time2 = time1 + timeinterval1 = ", YY, "/", MM, "/", DD, " ", &
         H, ":", M, ":", S
```

34.2.3 Time Comparison

This example shows how to compare two `ESMF_Times`.

```
if (time2 > time1) then
  print *, "time2 is larger than time1"
else
  print *, "time1 is smaller than or equal to time2"
endif

! finalize ESMF framework
call ESMF_Finalize(rc)


end program ESMF_TimeEx
```

34.3 Restrictions and Future Work

1. **Limits on size and resolution of Time.** The limits on the size and resolution of the time representation are based on the 64-bit and 32-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9223372036854775807. This corresponds to a maximum size of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

   For fractional seconds, a signed 32-bit integer will handle a resolution of +/- 2^{31}-1, or +/- 2,147,483,647 parts of a second.

34.4 Class API

34.4.1 ESMF_TimeOperator(+) - Increment a Time by a TimeInterval

INTERFACE:

```
interface operator(+)
time2 = time1 + timeinterval
```

RETURN VALUE:

```
type(ESMF_Time) :: time2
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

DESCRIPTION:

Overloads the (+) operator for the ESMF_Time class to increment time1 with timeinterval and return the result as an ESMF_Time.
The arguments are:

time1 The ESMF_Time to increment.

timeinterval The ESMF_TimeInterval to add to the given ESMF_Time.

34.4.2 ESMF_TimeOperator(-) - Decrement a Time by a TimeInterval

INTERFACE:

```
interface operator(-)
time2 = time1 - timeinterval
```

RETURN VALUE:

```
type(ESMF_Time) :: time2
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

DESCRIPTION:

Overloads the (-) operator for the ESMF_Time class to decrement time1 with timeinterval, and return the result as an ESMF_Time.
The arguments are:

time1 The ESMF_Time to decrement.

timeinterval The ESMF_TimeInterval to subtract from the given ESMF_Time.

34.4.3 ESMF_TimeOperator(-) - Return the difference between two Times

INTERFACE:

```
interface operator(-)
time3 = time1 - time2
```

RETURN VALUE:

```
type(ESMF_Time) :: time3
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_Time),         intent(in) :: time2
```

DESCRIPTION:

Overloads the (-) operator for the `ESMF_Time` class to return the difference between `time1` and `time2` as an `ESMF_TimeInterval`. It is assumed that `time1` is later than `time2`; if not, the resulting `ESMF_TimeInterval` will have a negative value.
The arguments are:

time1 The first `ESMF_Time` in comparison.

time2 The second `ESMF_Time` in comparison.

34.4.4 ESMF_TimeOperator(==) - Test if Time 1 is equal to Time 2

INTERFACE:

```
interface operator(==)
if (time1 == time2) then ... endif
             OR
result = (time1 == time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Time` class to return true if `time1` and `time2` are equal, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.5 ESMF_TimeOperator(/=) - Test if Time 1 is not equal to Time 2

INTERFACE:

```
interface operator(/=)
if (time1 /= time2) then ... endif
             OR
result = (time1 /= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Time` class to return true if `time1` and `time2` are not equal, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.6 ESMF_TimeOperator(<) - Test if Time 1 is less than Time 2

INTERFACE:

```
interface operator(<)
if (time1 < time2) then ... endif
             OR
result = (time1 < time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (<) operator for the `ESMF_Time` class to return true if `time1` is less than `time2`, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.7 ESMF_TimeOperator(<=) - Test if Time 1 is less than or equal to Time 2

INTERFACE:

```
interface operator(<=)
if (time1 <= time2) then ... endif
             OR
result = (time1 <= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (<=) operator for the `ESMF_Time` class to return true if `time1` is less than or equal to `time2`, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.8 ESMF_TimeOperator(>) - Test if Time 1 is greater than Time 2

INTERFACE:

```
interface operator(>)
if (time1 > time2) then ... endif
             OR
result = (time1 > time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (>) operator for the `ESMF_Time` class to return true if `time1` is greater than `time2`, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.9 ESMF_TimeOperator(>=) - Test if Time 1 is greater than or equal to Time 2

INTERFACE:

```
interface operator(>=)
if (time1 >= time2) then ... endif
             OR
result = (time1 >= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

DESCRIPTION:

Overloads the (>=) operator for the `ESMF_Time` class to return true if `time1` is greater than or equal to `time2`, and false otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

34.4.10 ESMF_TimeGet - Get a Time value

INTERFACE:

```
subroutine ESMF_TimeGet(time, yy, yy_i8, &
                              mm, dd, &
                              d, d_i8, &
                              h, m, &
                              s, s_i8, &
                              ms, us, ns, &
                              d_r8, h_r8, m_r8, s_r8, &
                              ms_r8, us_r8, ns_r8, &
                              sN, sD, &
                              calendar, calendarType, timeZone, &
                              timeString, dayOfWeek, midMonth, &
                              dayOfYear,  dayOfYear_r8, &
                              dayOfYear_intvl, rc)
```

ARGUMENTS:

```
type(ESMF_Time),          intent(in)            :: time
integer(ESMF_KIND_I4),   intent(out), optional :: yy
integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
integer,                  intent(out), optional :: mm
integer,                  intent(out), optional :: dd
integer(ESMF_KIND_I4),   intent(out), optional :: d
integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
```

```
integer(ESMF_KIND_I4),   intent(out), optional :: h
integer(ESMF_KIND_I4),   intent(out), optional :: m
integer(ESMF_KIND_I4),   intent(out), optional :: s
integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
integer(ESMF_KIND_I4),   intent(out), optional :: ms
integer(ESMF_KIND_I4),   intent(out), optional :: us
integer(ESMF_KIND_I4),   intent(out), optional :: ns
real(ESMF_KIND_R8),      intent(out), optional :: d_r8
real(ESMF_KIND_R8),      intent(out), optional :: h_r8
real(ESMF_KIND_R8),      intent(out), optional :: m_r8
real(ESMF_KIND_R8),      intent(out), optional :: s_r8
real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
real(ESMF_KIND_R8),      intent(out), optional :: us_r8
real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
integer(ESMF_KIND_I4),   intent(out), optional :: sN    ! not implemented
integer(ESMF_KIND_I4),   intent(out), optional :: sD    ! not implemented
type(ESMF_Calendar),     intent(out), optional :: calendar
type(ESMF_CalendarType), intent(out), optional :: calendarType
integer,                 intent(out), optional :: timeZone
character (len=*),       intent(out), optional :: timeString
integer,                 intent(out), optional :: dayOfWeek
type(ESMF_Time),         intent(out), optional :: midMonth
integer(ESMF_KIND_I4),   intent(out), optional :: dayOfYear
real(ESMF_KIND_R8),      intent(out), optional :: dayOfYear_r8 ! not implemented
type(ESMF_TimeInterval), intent(out), optional :: dayOfYear_intvl
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Gets the value of `time` in units specified by the user via Fortran optional arguments. See `ESMF_TimeSet()` above for a description of time units and calendars.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers. For example, if a time value is 5 and 3/8 seconds (s=5, sN=3, sD=8), and you want to get it as floating point seconds, you would get 5.375 (s_r8=5.375). (Fractions and reals not implemented yet).
Units are bound (normalized) to the next larger unit specified. For example, if a time is defined to be 2:00 am on a particular date, then `ESMF_TimeGet(h = hours, s = seconds)` would return `hours = 2`, `seconds = 0`, whereas `ESMF_TimeGet(s=seconds)` would return `seconds = 7200`.
For `timeString`, `dayOfWeek`, `midMonth`, `dayOfYear`, `dayOfYear_r8`, and `dayOfYear_intvl` described below, valid calendars are Gregorian, Julian Date, No Leap, 360 Day and Custom calendars. Not valid for Julian day or no calendar.

For timeString, convert `ESMF_Time`'s value into ISO 8601 format YYYY-MM-DDThh:mm:ss. See [1].
For dayOfWeek, gets the day of the week the given `ESMF_Time` instant falls on. ISO 8601 standard: Monday = 1 through Sunday = 7. See [1].
For midMonth, gets the middle time instant of the month that the given `ESMF_Time` instant falls on.
For dayOfYear, gets the day of the year that the given `ESMF_Time` instant falls on. See range discusion in argument list below. Return as an integer value.
For dayOfYear_r8, gets the day of the year the given `ESMF_Time` instant falls on. See range discusion in argument list below. Return as floating point value; fractional part represents the time of day. (Fractions and reals not implemented yet).
For dayOfYear_intvl, gets the day of the year the given `ESMF_Time` instant falls on. Return as an `ESMF_TimeInterval`.
The arguments are:

time The object instance to query.

[yy] Integer year (>= 32-bit).

[yy_i8] Integer year (large, >= 64-bit).

[mm] Integer month.

[dd] Integer day of the month.

[d] Integer Julian days (>= 32-bit).

[d_i8] Integer Julian days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

[s_i8] Integer seconds (large, >= 64-bit).

[ms] Integer milliseconds. (Not implemented yet).

[us] Integer microseconds. (Not implemented yet).

[ns] Integer nanoseconds. (Not implemented yet).

[d_r8] Double precision days. (Not implemented yet).

[h_r8] Double precision hours. (Not implemented yet).

[m_r8] Double precision minutes. (Not implemented yet).

[s_r8] Double precision seconds. (Not implemented yet).

[ms_r8] Double precision milliseconds. (Not implemented yet).

[us_r8] Double precision microseconds. (Not implemented yet).

[ns_r8] Double precision nanoseconds. (Not implemented yet).

[sN] Integer numerator portion of fractional seconds (sN/sD). (Not implemented yet).

[sD] Integer denominator portion of fractional seconds (sN/sD). (Not implemented yet).

[calendar] Associated `Calendar`.

[calendarType] Associated `CalendarType`.

[timeZone] Associated timezone (hours offset from UCT, e.g. EST = -5). (Not implemented yet).

[timeString] Convert time value to ISO 8601 format string YYYY-MM-DDThh:mm:ss. See [1].

[dayOfWeek] The time instant's day of the week [1-7].

[MidMonth] The given time instant's middle-of-the-month time instant.

[dayOfYear] The `ESMF_Time` instant's integer day of the year. [1-366] for Gregorian and Julian calendars, [1-365] for No-Leap calendar. [1-360] for 360-Day calendar. User-defined range for Custom calendar.

[dayOfYear_r8] The `ESMF_Time` instant's floating point day of the year. [1.x-366.x] for Gregorian and Julian calendars, [1.x-365.x] for No-Leap calendar. [1.x-360.x] for 360-Day calendar. User-defined range for Custom calendar. (Not implemented yet).

[dayOfYear_intvl] The `ESMF_Time` instant's day of the year as an `ESMF_TimeInterval`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

34.4.11 ESMF_TimeIsSameCalendar - Compare Calendars of two Times

INTERFACE:

```
function ESMF_TimeIsSameCalendar(time1, time2, rc)
```

RETURN VALUE:

```
logical :: ESMF_TimeIsSameCalendar
```

ARGUMENTS:

```
type(ESMF_Time), intent(in)            :: time1
type(ESMF_Time), intent(in)            :: time2
integer,         intent(out), optional :: rc
```

DESCRIPTION:

Returns true if the Calendars in these Times are the same, false otherwise.
The arguments are:

time1 The first `ESMF_Time` in comparison.

time2 The second `ESMF_Time` in comparison.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

34.4.12 ESMF_TimePrint - Print the contents of a Time

INTERFACE:

```
subroutine ESMF_TimePrint(time, options, rc)
```

ARGUMENTS:

```
type(ESMF_Time),   intent(in)            :: time
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Prints out the contents of an `ESMF_Time` to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.
The arguments are:

time The `ESMF_Time` to be printed out.

[options] Print options. If none specified, prints all Time property values.
"string" - prints Time's value in ISO 8601 format YYYY-MM-DDThh:mm:ss. See [1].

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

34.4.13 ESMF_TimeSet - Initialize or set a Time

INTERFACE:

```
subroutine ESMF_TimeSet(time, yy, yy_i8, &
                              mm, dd, &
                              d, d_i8, &
                              h, m, &
                              s, s_i8, &
                              ms, us, ns, &
                              d_r8, h_r8, m_r8, s_r8, &
                              ms_r8, us_r8, ns_r8, &
                              sN, sD, calendar, calendarType, &
                              timeZone, rc)
```

ARGUMENTS:

```
type(ESMF_Time),          intent(inout)          :: time
integer(ESMF_KIND_I4),    intent(in),  optional :: yy
integer(ESMF_KIND_I8),    intent(in),  optional :: yy_i8
integer,                  intent(in),  optional :: mm
integer,                  intent(in),  optional :: dd
integer(ESMF_KIND_I4),    intent(in),  optional :: d
integer(ESMF_KIND_I8),    intent(in),  optional :: d_i8
integer(ESMF_KIND_I4),    intent(in),  optional :: h
integer(ESMF_KIND_I4),    intent(in),  optional :: m
integer(ESMF_KIND_I4),    intent(in),  optional :: s
integer(ESMF_KIND_I8),    intent(in),  optional :: s_i8
integer(ESMF_KIND_I4),    intent(in),  optional :: ms    ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: us    ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: ns    ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: d_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: h_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: m_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: s_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: ms_r8 ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: us_r8 ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: ns_r8 ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: sN    ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: sD    ! not implemented
type(ESMF_Calendar),      intent(in),  optional :: calendar
type(ESMF_CalendarType), intent(in),  optional :: calendarType
integer,                  intent(in),  optional :: timeZone ! not implemented
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Initializes an `ESMF_Time` with a set of user-specified units via Fortran optional arguments.
The range of valid values for mm and dd depend on the calendar used. For Gregorian, Julian, and No-Leap calendars, mm is [1-12] and dd is [1-28,29,30, or 31], depending on the value of mm and whether yy or yy_i8 is a leap year. For the 360-day calendar, mm is [1-12] and dd is [1-30]. For the Julian-day and No-calendar, yy, yy_i8, mm, and dd are invalid inputs, since these calendars do not define them. When valid, the yy and yy_i8 arguments should be fully specified, e.g. 2003 instead of 03. yy and yy_i8 ranges are only limited by machine word size, except for the Gregorian calendar, where the lower year limit is -4800. This is a limitation of the Gregorian date-to-Julian day

conversion algorithm used to convert a Gregorian date to the internal representation of seconds. The algorithm is from Henry F. Fliegel and Thomas C. Van Flandern, in Communications of the ACM, CACM, volume 11, number 10, October 1968, p. 657. The Custom calendar will have a user-defined definition of yy, yy_i8, mm, and dd.
The Julian day specifier, d or d_i8, can be used with either the Julian-day or No-calendar, and has a valid range depending on the word size. For a 32-bit d, the range is [-24855 to 24855]. For a 64-bit d or d_i8, the valid range is [-32045 to 1,067,519,911]. The reference day of d=0 corresponds to 11/24/-4713 in the Gregorian calendar, which is derived from the Julian-day to Gregorian conversion algorithm by Fliegel/Van Flandern (see above). The lower range day value of -32045 is the lowest for which the Fliegel/Van Flandern algorithm works. The upper range of the algorithm is only limited by machine word size.
Hours, minutes, seconds, and sub-seconds can be used with any calendar.
Time manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers. Sub-second values will be represented internally with an integer numerator and denominator fraction (sN/sD). The smallest resolution will be nanoseconds (denominator), as per Time Manager requirement TMG3.1. Anything smaller will be truncated. For example, pi would be represented as s=3, sN=141592654, sD=1000000000. (Fractions and reals not implemented yet).
The arguments are:

time The object instance to initialize.

[yy] Integer year (>= 32-bit). Default = 0

[yy_i8] Integer year (large, >= 64-bit). Default = 0

[mm] Integer month. Default = 1

[dd] Integer day of the month. Default = 1

[d] Integer Julian days (>= 32-bit). Default = 0

[d_i8] Integer Julian days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0. (Not implemented yet).

[us] Integer microseconds. Default = 0. (Not implemented yet).

[ns] Integer nanoseconds. Default = 0. (Not implemented yet).

[d_r8] Double precision days. Default = 0.0. (Not implemented yet).

[h_r8] Double precision hours. Default = 0.0. (Not implemented yet).

[m_r8] Double precision minutes. Default = 0.0. (Not implemented yet).

[s_r8] Double precision seconds. Default = 0.0. (Not implemented yet).

[ms_r8] Double precision milliseconds. Default = 0.0. (Not implemented yet).

[us_r8] Double precision microseconds. Default = 0.0. (Not implemented yet).

[ns_r8] Double precision nanoseconds. Default = 0.0. (Not implemented yet).

[sN] Integer numerator portion of fractional seconds (sN/sD). Default = 0. (Not implemented yet).

[sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1. (Not implemented yet).

calendar Associated Calendar. Defaults to calendar ESMF_CAL_NOCALENDAR or default specified in ESMF_Initialize() or ESMF_CalendarSetDefault(). Alternate to, and mutually exclusive with, calendarType below. Primarily for specifying a custom calendar type.

[calendarType] Alternate to, and mutually exclusive with, calendar above. More convenient way of specifying a built-in calendar type.

[timeZone] Associated timezone (hours offset from UTC, e.g. EST = -5). Default = 0 (UTC). (Not implemented yet).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

34.4.14 ESMF_TimeSyncToRealTime - Get system real time (wall clock time)

INTERFACE:

```
subroutine ESMF_TimeSyncToRealTime(time, rc)
```

ARGUMENTS:

```
type(ESMF_Time), intent(inout) :: time
integer, intent(out), optional :: rc
```

DESCRIPTION:

Gets the system real time (wall clock time), and returns it as an ESMF_Time. Accurate to the nearest second.
The arguments are:

time The object instance to receive the real time.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

34.4.15 ESMF_TimeValidate - Validate a Time

INTERFACE:

```
subroutine ESMF_TimeValidate(time, options, rc)
```

ARGUMENTS:

```
type(ESMF_Time),    intent(in)            :: time
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Checks whether an ESMF_Time is valid. The options control the type of validation.
The arguments are:

time ESMF_Time instant to be validated.

[options] Validation options. If none specified, validates all time property values.
"calendar" - validate only the time's calendar.
"timezone" - validate only the time's timezone.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

35 TimeInterval Class

35.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative. Like the Time interface, the TimeInterval interface is designed so that you can choose one or more options from a list of time units in order to specify a TimeInterval. See Section 34.1, Table 1 for the available options.
There are TimeInterval methods defined for setting and getting a TimeInterval, for incrementing and decrementing a TimeInterval by another TimeInterval, and for multiplying and dividing TimeIntervals by integers, reals, fractions and other TimeIntervals. Methods are also defined to take the absolute value and negative absolute value of a TimeInterval, and for comparing the length of two TimeIntervals.
The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. For example, the difference between two Times is a TimeInterval.
When a TimeInterval is used in calculations that involve an absolute reference time, such as incrementing a Time with a TimeInterval, calendar dependencies may be introduced. The length of the time period that the TimeInterval represents will depend on the reference Time and the standard calendar that is associated with it. The calendar dependency becomes apparent when, for example, adding a TimeInterval of 1 day to the Time of February 28, 1996, at 4:00pm EST. In a 360 day calendar, the resulting date would be February 29, 1996, at 4:00pm EST. In a no-leap calendar, the result would be March 1, 1996, at 4:00pm EST.
TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks (Section 36.1) and Alarms (Section 37.1).

35.2 Use and Examples

A typical use for a TimeInterval in a geophysical model is representation of the time step by which the model is advanced. Some models change the size of their time step as the model run progresses; this could be done by incrementing or decrementing the original time step by another TimeInterval, or by dividing or multiplying the time step by an integer value. An example of advancing model time using a TimeInterval representation of a time step is shown in Section 36.1.
The following brief example shows how to create, initialize and manipulate `TimeInterval`.

```
! !PROGRAM: ESMF_TimeIntervalEx - Time Interval initialization and manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Time Interval initialization and manipulation
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate some time intervals
      type(ESMF_TimeInterval) :: timeinterval1, timeinterval2, timeinterval3

      ! local variables
      integer :: d, h, m, s

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultCalendar=ESMF_CAL_GREGORIAN, rc=rc)
```

35.2.1 Time Interval Initialization

This example shows how to initialize two `ESMF_TimeIntervals`.

```
! initialize time interval1 to 1 day
call ESMF_TimeIntervalSet(timeinterval1, d=1, rc=rc)


call ESMF_TimeIntervalPrint(timeinterval1, "string", rc)


! initialize time interval2 to 4 days, 1 hour, 30 minutes, 10 seconds
call ESMF_TimeIntervalSet(timeinterval2, d=4, h=1, m=30, s=10, rc=rc)


call ESMF_TimeIntervalPrint(timeinterval2, "string", rc)
```

35.2.2 Time Interval Conversion

This example shows how to convert `ESMF_TimeIntervals` into different units.

```
call ESMF_TimeIntervalGet(timeinterval1, s=s, rc=rc)
print *, "Time Interval1 = ", s, " seconds."


call ESMF_TimeIntervalGet(timeinterval2, h=h, m=m, s=s, rc=rc)
print *, "Time Interval2 = ", h, " hours, ", m, " minutes, ", &
                              s, " seconds."
```

35.2.3 Time Interval Difference

This example shows how to calculate the difference between two `ESMF_TimeIntervals`.

```
! difference between two time intervals
timeinterval3 = timeinterval2 - timeinterval1
call ESMF_TimeIntervalGet(timeinterval3, d=d, h=h, m=m, s=s, rc=rc)
print *, "Difference between TimeInterval2 and TimeInterval1 = ", &
         d, " days, ", h, " hours, ", m, " minutes, ", s, " seconds."
```

35.2.4 Time Interval Multiplication

This example shows how to multiply an `ESMF_TimeInterval`.

```
! multiply time interval by an integer
timeinterval3 = timeinterval2 * 3
call ESMF_TimeIntervalGet(timeinterval3, d=d, h=h, m=m, s=s, rc=rc)
print *, "TimeInterval2 multiplied by 3 = ", d, " days, ", h, &
         " hours, ", m, " minutes, ", s, " seconds."
```

35.2.5 Time Interval Comparison

This example shows how to compare two `ESMF_TimeIntervals`.

```
! comparison
if (timeinterval1 < timeinterval2) then
  print *, "TimeInterval1 is smaller than TimeInterval2"
else
  print *, "TimeInterval1 is larger than or equal to TimeInterval2"
end if

! finalize ESMF framework
call ESMF_Finalize(rc)


end program ESMF_TimeIntervalEx
```

35.3 Restrictions and Future Work

1. **Limits on time span.** The limits on the time span that can be represented are based on the 64-bit and 32-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9223372036854775807. This corresponds to a range of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

35.4 Class API

35.4.1 ESMF_TimeIntervalOperator(+) - Add two TimeIntervals

INTERFACE:

```
interface operator(+)
sum = timeinterval1 + timeinterval2
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: sum
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (+) operator for the `ESMF_TimeInterval` class to add `timeinterval1` to `timeinterval2` and return the sum as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The augend.

timeinterval2 The addend.

35.4.2 ESMF_TimeIntervalOperator(-) - Subtract one TimeInterval from another

INTERFACE:

```
interface operator(-)
difference = timeinterval1 - timeinterval2
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: difference
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (-) operator for the `ESMF_TimeInterval` class to subtract `timeinterval2` from `timeinterval1` and return the difference as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The minuend.

timeinterval2 The subtrahend.

35.4.3 ESMF_TimeIntervalOperator(/) - Divide two TimeIntervals, return double precision quotient

INTERFACE:

```
interface operator(/)
quotient = timeinterval1 / timeinterval2
```

RETURN VALUE:

```
real(ESMF_KIND_R8) :: quotient
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (/) operator for the `ESMF_TimeInterval` class to return `timeinterval1` divided by `timeinterval2` as a double precision quotient.
The arguments are:

timeinterval1 The dividend.

timeinterval2 The divisor.

35.4.4 ESMF_TimeIntervalFunction(MOD) - Divide two TimeIntervals, return TimeInterval remainder

INTERFACE:

```
interface MOD
remainder = MOD(timeinterval1, timeinterval2)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: remainder
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the pre-defined MOD() function for the `ESMF_TimeInterval` class to return the remainder of `timeinterval1` divided by `timeinterval2` as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The dividend.

timeinterval2 The divisor.

35.4.5 ESMF_TimeIntervalOperator(x) - Multiply a TimeInterval by an integer

INTERFACE:

```
interface operator(*)
product = timeinterval * multiplier
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: product
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
integer(ESMF_KIND_I4),   intent(in) :: multiplier
```

DESCRIPTION:

Overloads the (*) operator for the `ESMF_TimeInterval` class to multiply a `timeinterval` by an integer `multiplier`, and return the product as an `ESMF_TimeInterval`.
Commutative complement to overloaded operator (*) below.
The arguments are:

timeinterval The multiplicand.

mutliplier The integer multiplier.

35.4.6 ESMF_TimeIntervalOperator(x) - Multiply a TimeInterval by an integer

INTERFACE:

```
interface operator(*)
product = multiplier * timeinterval
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: product
```

ARGUMENTS:

```
integer(ESMF_KIND_I4),   intent(in) :: multiplier
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

DESCRIPTION:

Overloads the (*) operator for the `ESMF_TimeInterval` class to multiply a `timeinterval` by an integer `multiplier`, and return the product as an `ESMF_TimeInterval`.
Commutative complement to overloaded operator (*) above.
The arguments are:

mutliplier The integer multiplier.

timeinterval The multiplicand.

35.4.7 ESMF_TimeIntervalOperator(==) - Test if TimeInterval 1 is equal to TimeInterval 2

INTERFACE:

```
interface operator(==)
if (timeinterval1 == timeinterval2) then ... endif
             OR
result = (timeinterval1 == timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` and `timeinterval2` are equal, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.8 ESMF_TimeIntervalOperator(/=) - Test if TimeInterval 1 is not equal to TimeInterval 2

INTERFACE:

```
interface operator(/=)
if (timeinterval1 /= timeinterval2) then ... endif
             OR
result = (timeinterval1 /= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` and `timeinterval2` are not equal, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.9 ESMF_TimeIntervalOperator(<) - Test if TimeInterval 1 is less than TimeInterval 2

INTERFACE:

```
interface operator(<)
if (timeinterval1 < timeinterval2) then ... endif
             OR
result = (timeinterval1 < timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (<) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` is less than `timeinterval2`, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.10 ESMF_TimeIntervalOperator(<=) - Test if TimeInterval 1 is less than or equal to TimeInterval 2

INTERFACE:

```
interface operator(<=)
if (timeinterval1 <= timeinterval2) then ... endif
             OR
result = (timeinterval1 <= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (<=) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` is less than or equal to `timeinterval2`, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.11 ESMF_TimeIntervalOperator(>) - Test if TimeInterval 1 is greater than TimeInterval 2

INTERFACE:

```
interface operator(>)
if (timeinterval1 > timeinterval2) then ... endif
             OR
result = (timeinterval1 > timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (<) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` is greater than `timeinterval2`, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.12 ESMF_TimeIntervalOperator(>=) - Test if TimeInterval 1 is greater than or equal to TimeInterval 2

INTERFACE:

```
interface operator(>=)
if (timeinterval1 >= timeinterval2) then ... endif
             OR
result = (timeinterval1 >= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

DESCRIPTION:

Overloads the (<=) operator for the `ESMF_TimeInterval` class to return true if `timeinterval1` is greater than or equal to `timeinterval2`, and false otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

35.4.13 ESMF_TimeIntervalAbsValue - Get the absolute value of a TimeInterval

INTERFACE:

```
function ESMF_TimeIntervalAbsValue(timeinterval)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: ESMF_TimeIntervalAbsValue
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

DESCRIPTION:

Returns the absolute value of `timeinterval`.
The argument is:

timeinterval The object instance to take the absolute value of. Absolute value is returned as the value of the function.

35.4.14 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
      subroutine ESMF_TimeIntervalGet(timeinterval, &
                                      yy, yy_i8, &
                                      mm, mm_i8, &
                                      d, d_i8, &
                                      h, m, &
                                      s, s_i8, &
                                      ms, us, ns, &
                                      d_r8, h_r8, m_r8, s_r8, &
                                      ms_r8, us_r8, ns_r8, &
                                      sN, sD, &
                                      startTime, endTime, &
                                      calendar, calendarType, &
                                      startTimeIn, endTimeIn, &
                                      calendarIn, calendarTypeIn, &
                                      timeString, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeinterval
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: mm
      integer(ESMF_KIND_I8),   intent(out), optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms    ! not implemented
      integer(ESMF_KIND_I4),   intent(out), optional :: us    ! not implemented
      integer(ESMF_KIND_I4),   intent(out), optional :: ns    ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8  ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: h_r8  ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: m_r8  ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: s_r8  ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: ms_r8 ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: us_r8 ! not implemented
      real(ESMF_KIND_R8),      intent(out), optional :: ns_r8 ! not implemented
      integer(ESMF_KIND_I4),   intent(out), optional :: sN    ! not implemented
      integer(ESMF_KIND_I4),   intent(out), optional :: sD    ! not implemented
      type(ESMF_Time),         intent(out), optional :: startTime
      type(ESMF_Time),         intent(out), optional :: endTime
      type(ESMF_Calendar),     intent(out), optional :: calendar
      type(ESMF_CalendarType), intent(out), optional :: calendarType
      type(ESMF_Time),         intent(in),  optional :: startTimeIn    ! Input
      type(ESMF_Time),         intent(in),  optional :: endTimeIn      ! Input
      type(ESMF_Calendar),     intent(in),  optional :: calendarIn     ! Input
      type(ESMF_CalendarType), intent(in),  optional :: calendarTypeIn ! Input
      character (len=*),       intent(out), optional :: timeString
      integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Gets the value of `timeinterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers. (Fractions and reals not implemented yet).
Units are bound (normalized) to the next larger unit specified. For example, if a time interval is defined to be one day, then `ESMF_TimeIntervalGet(d = days, s = seconds)` would return `days = 1`, `seconds = 0`, whereas `ESMF_TimeIntervalGet(s = seconds)` would return `seconds = 86400`.
See `../include/ESMC_BaseTime.h` and `../include/ESMC_TimeInterval.h` for complete description.
For timeString, converts `ESMF_TimeInterval`'s value into ISO 8601 format PyYmMdDThHmMsS. See [1].
The arguments are:

timeinterval The object instance to query.

[yy] Integer years (>= 32-bit).

[yy_i8] Integer years (large, >= 64-bit).

[mm] Integer months (>= 32-bit).

[mm_i8] Integer months (large, >= 64-bit).

[d] Integer Julian days (>= 32-bit).

[d_i8] Integer Julian days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

[s_i8] Integer seconds (large, >= 64-bit).

[ms] Integer milliseconds. (Not implemented yet).

[us] Integer microseconds. (Not implemented yet).

[ns] Integer nanoseconds. (Not implemented yet).

[d_r8] Double precision days. (Not implemented yet).

[h_r8] Double precision hours. (Not implemented yet).

[m_r8] Double precision minutes. (Not implemented yet).

[s_r8] Double precision seconds. (Not implemented yet).

[ms_r8] Double precision milliseconds. (Not implemented yet).

[us_r8] Double precision microseconds. (Not implemented yet).

[ns_r8] Double precision nanoseconds. (Not implemented yet).

[sN] Integer numerator portion of fractional seconds (sN/sD). (Not implemented yet).

[sD] Integer denominator portion of fractional seconds (sN/sD). (Not implemented yet).

[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

[endTime] Ending time, if set, of an absolute calendar interval (yy, mm, and/or d).

[calendar] Associated `Calendar`, if any.

[calendarType] Associated `CalendarType`, if any.

[startTimeIn] INPUT argument: pins a calendar interval to a specific point in time to allow conversion between relative units (yy, mm, d) and absolute units (d, h, m, s). Overrides any startTime and/or endTime previously set. Mutually exclusive with endTimeIn and calendarIn.

[endTimeIn] INPUT argument: pins a calendar interval to a specific point in time to allow conversion between relative units (yy, mm, d) and absolute units (d, h, m, s). Overrides any startTime and/or endTime previously set. Mutually exclusive with startTimeIn and calendarIn.

[calendarIn] INPUT argument: pins a calendar interval to a specific calendar to allow conversion between relative units (yy, mm, d) and absolute units (d, h, m, s). Mutually exclusive with startTimeIn and endTimeIn since they contain a calendar. Alternate to, and mutually exclusive with, calendarTypeIn below. Primarily for specifying a custom calendar type.

[calendarTypeIn] INPUT argument: Alternate to, and mutually exclusive with, calendarIn above. More convenient way of specifying a built-in calendar type.

timeString The string to return.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

35.4.15 ESMF_TimeIntervalNegAbsValue - Get the negative absolute value of a TimeInterval

INTERFACE:

```
function ESMF_TimeIntervalNegAbsValue(timeinterval)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: ESMF_TimeIntervalNegAbsValue
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

DESCRIPTION:

Returns the negative absolute value of `timeinterval`.
The argument is:

timeinterval The object instance to take the negative absolute value of. Negative absolute value is returned as the value of the function.

35.4.16 ESMF_TimeIntervalPrint - Print the contents of a TimeInterval

INTERFACE:

```
subroutine ESMF_TimeIntervalPrint(timeinterval, options, rc)
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in)            :: timeinterval
character (len=*),       intent(in),  optional :: options
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Prints out the contents of an `ESMF_TimeInterval` to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.
The arguments are:

timeinterval Time interval to be printed out.

[options] Print options. If none specified, prints all `timeinterval` property values.
"string" - prints `timeinterval`'s value in ISO 8601 format PyYmMdDThHmMsS. See [1].

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

35.4.17 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
subroutine ESMF_TimeIntervalSet(timeinterval, &
                                 yy, yy_i8, &
                                 mm, mm_i8, &
                                 d, d_i8, &
                                 h, m, &
                                 s, s_i8, &
                                 ms, us, ns, &
                                 d_r8, h_r8, m_r8, s_r8, &
                                 ms_r8, us_r8, ns_r8, &
                                 sN, sD, &
                                 startTime, endTime, &
                                 calendar, calendarType, rc)
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(inout)         :: timeinterval
integer(ESMF_KIND_I4),   intent(in),  optional :: yy
integer(ESMF_KIND_I8),   intent(in),  optional :: yy_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: mm
integer(ESMF_KIND_I8),   intent(in),  optional :: mm_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: d
integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: h
integer(ESMF_KIND_I4),   intent(in),  optional :: m
integer(ESMF_KIND_I4),   intent(in),  optional :: s
integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: ms    ! not implemented
integer(ESMF_KIND_I4),   intent(in),  optional :: us    ! not implemented
integer(ESMF_KIND_I4),   intent(in),  optional :: ns    ! not implemented
```

```
real(ESMF_KIND_R8),       intent(in),  optional :: d_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: h_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: m_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: s_r8  ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: ms_r8 ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: us_r8 ! not implemented
real(ESMF_KIND_R8),       intent(in),  optional :: ns_r8 ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: sN    ! not implemented
integer(ESMF_KIND_I4),    intent(in),  optional :: sD    ! not implemented
type(ESMF_Time),          intent(in),  optional :: startTime
type(ESMF_Time),          intent(in),  optional :: endTime
type(ESMF_Calendar),      intent(in),  optional :: calendar
type(ESMF_CalendarType),  intent(in),  optional :: calendarType
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Sets the value of the `ESMF_TimeInterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers. (Fractions and reals not implemented yet).
Ranges are limited only by machine word size. Numeric defaults are 0, except for sD, which is 1.
The arguments are:

timeinterval The object instance to initialize.

[yy] Integer years (>= 32-bit). Default = 0

[yy_i8] Integer years (large, >= 64-bit). Default = 0

[mm] Integer months (>= 32-bit). Default = 0

[mm_i8] Integer months (large, >= 64-bit). Default = 0

[d] Integer Julian days (>= 32-bit). Default = 0

[d_i8] Integer Julian days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0. (Not implemented yet).

[us] Integer microseconds. Default = 0. (Not implemented yet).

[ns] Integer nanoseconds. Default = 0. (Not implemented yet).

[d_r8] Double precision days. Default = 0.0. (Not implemented yet).

[h_r8] Double precision hours. Default = 0.0. (Not implemented yet).

[m_r8] Double precision minutes. Default = 0.0. (Not implemented yet).

[s_r8] Double precision seconds. Default = 0.0. (Not implemented yet).

[ms_r8] Double precision milliseconds. Default = 0.0. (Not implemented yet).

[us_r8] Double precision microseconds. Default = 0.0. (Not implemented yet).

[ns_r8] Double precision nanoseconds. Default = 0.0. (Not implemented yet).

[sN] Integer numerator portion of fractional seconds (sN/sD). Default = 0. (Not implemented yet).

[sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1. (Not implemented yet).

[startTime] Starting time of an absolute calendar interval (yy, mm, and/or d); pins a calendar interval to a specific point in time. If not set, and endTime and calendar also not set, calendar interval "floats" across all calendars and times. Mutually exclusive with calendar since it contains a calendar.

[endTime] Ending time of an absolute calendar interval (yy, mm, and/or d); pins a calendar interval to a specific point in time. If not set, and startTime and calendar also not set, calendar interval "floats" across all calendars and times. Mutually exclusive with calendar since it contains a calendar.

[calendar] `Calendar` used to give better definition to calendar interval (yy, mm, and/or d) for arithmetic, comparison, and conversion operations. Allows calendar interval to "float" across all times on a specific calendar. Default = NULL; if startTime and endTime also not specified, calendar interval "floats" across all calendars and times. Mutually exclusive with startTime and endTime since they contain a calendar. Alternate to, and mutually exclusive with, calendarType below. Primarily for specifying a custom calendar type.

[calendarType] Alternate to, and mutually exclusive with, calendar above. More convenient way of specifying a built-in calendar type.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

35.4.18 ESMF_TimeIntervalValidate - Validate a TimeInterval

INTERFACE:

```
subroutine ESMF_TimeIntervalValidate(timeinterval, options, rc)
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in)            :: timeinterval
character (len=*),       intent(in),  optional :: options
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Checks whether a `timeinterval` is valid. The options control the type of validation.
The arguments are:

timeinterval `ESMF_TimeInterval` to be validated.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36 Clock Class

36.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, previous time, and a time step. It can also store a reference time, typically the time instant at which a simulation originally began. For a restart run, the reference time can be different than the start time, when the application execution resumes.
A user can call the `ESMF_ClockSet` method and reset the time step as desired.
A Clock also stores a list of Alarms, which can be set to flag events that occur at a specified time instant or at a specified time interval. See Section 37.1 for details on how to use Alarms.
There are methods for setting and getting the Times and Alarms associated with a Clock. Methods are defined for advancing the Clock's current time, checking if the stop time has been reached, and synchronizing with a real clock.

36.2 Use and Examples

The following is a typical sequence for using a Clock in a geophysical model.
At initialize:

- Set a Calendar.
- Set start time, stop time and time step as Times and Time Intervals.
- Create and Initialize a Clock using the start time, stop time and time step.
- Define Times and Time Intervals associated with special events, and use these to set Alarms.

At run:

- Advance the Clock, checking for ringing alarms as needed.
- Check if it is time to stop.

At finalize:

- Since Clocks and Alarms are deep classes, they need to be explicitly destroyed at finalization. Times and TimeIntervals are lightweight classes, so they don't need explicit destruction.

The following code example illustrates Clock usage.

```
! !PROGRAM: ESMF_ClockEx - Clock initialization and time-stepping
!
! !DESCRIPTION:
!
! This program shows an example of how to create, initialize, advance, and
! examine a basic clock
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate a clock
      type(ESMF_Clock) :: clock

      ! instantiate time_step, start and stop times
      type(ESMF_TimeInterval) :: timeStep
      type(ESMF_Time) :: startTime
      type(ESMF_Time) :: stopTime
```

```
! local variables for Get methods
type(ESMF_Time) :: currTime
integer(ESMF_KIND_I8) :: advanceCount
integer :: YY, MM, DD, H, M, S

! return code
integer :: rc


! initialize ESMF framework
call ESMF_Initialize(defaultCalendar=ESMF_CAL_GREGORIAN, rc=rc)
```

36.2.1 Clock Creation

This example shows how to create and initialize an `ESMF_Clock`.

```
! initialize time interval to 2 days, 4 hours (6 timesteps in 13 days)
call ESMF_TimeIntervalSet(timeStep, d=2, h=4, rc=rc)


! initialize start time to 4/1/2003 2:24:00 ( 1/10 of a day )
call ESMF_TimeSet(startTime, yy=2003, mm=4, dd=1, h=2, m=24, rc=rc)


! initialize stop time to 4/14/2003 2:24:00 ( 1/10 of a day )
call ESMF_TimeSet(stopTime, yy=2003, mm=4, dd=14, h=2, m=24, rc=rc)


! initialize the clock with the above values
clock = ESMF_ClockCreate("Clock 1", timeStep, startTime, stopTime, rc=rc)
```

36.2.2 Clock Advance

This example shows how to time-step an `ESMF_Clock`.

```
! time step clock from start time to stop time
do while (.not.ESMF_ClockIsStopTime(clock, rc))


  call ESMF_ClockPrint(clock, "currTime string", rc)


  call ESMF_ClockAdvance(clock, rc=rc)


end do
```

36.2.3 Clock Examination

This example shows how to examine an `ESMF_Clock`.

```
! get the clock's final current time
call ESMF_ClockGet(clock, currTime=currTime, rc=rc)
```

```
call ESMF_TimeGet(currTime, yy=YY, mm=MM, dd=DD, h=H, m=M, s=S, rc=rc)
print *, "The clock's final current time is ", YY, "/", MM, "/", DD, &
         " ", H, ":", M, ":", S

! get the number of times the clock was advanced
call ESMF_ClockGet(clock, advanceCount=advanceCount, rc=rc)
print *, "The clock was advanced ", advanceCount, " times."
```

36.2.4 Clock Destruction

This example shows how to destroy an `ESMF_Clock`.

```
! destroy clock
call ESMF_ClockDestroy(clock, rc)

! finalize ESMF framework
call ESMF_Finalize(rc)

end program ESMF_ClockEx
```

36.3 Restrictions and Future Work

1. **Limit on number of Alarms.** The maximum number of Alarms that can be associated with a Clock is currently hard-coded at 100. This is done in both Fortran and C++ with a #define for ease of modification.

36.4 Class API

36.4.1 ESMF_ClockOperator(==) - Test if Clock 1 is equal to Clock 2

INTERFACE:

```
interface operator(==)
if (clock1 == clock2) then ... endif
             OR
result = (clock1 == clock2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in) :: clock1
type(ESMF_Clock), intent(in) :: clock2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Clock` class. Compare two clocks for equality; return true if equal, false otherwise. Comparison is based on IDs, which are distinct for newly created clocks and identical for clocks created as copies.
The arguments are:

clock1 The first `ESMF_Clock` in comparison.

clock2 The second `ESMF_Clock` in comparison.

36.4.2 ESMF_ClockOperator(/=) - Test if Clock 1 is not equal to Clock 2

INTERFACE:

```
interface operator(/=)
if (clock1 /= clock2) then ... endif
             OR
result = (clock1 /= clock2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in) :: clock1
type(ESMF_Clock), intent(in) :: clock2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Clock` class. Compare two clocks for inequality; return true if not equal, false otherwise. Comparison is based on IDs, which are distinct for newly created clocks and identical for clocks created as copies.
The arguments are:

clock1 The first `ESMF_Clock` in comparison.

clock2 The second `ESMF_Clock` in comparison.

36.4.3 ESMF_ClockAdvance - Advance a Clock's current time by one time step

INTERFACE:

```
subroutine ESMF_ClockAdvance(clock, timeStep, ringingAlarmList, &
                             ringingAlarmCount, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),               intent(inout)          :: clock
type(ESMF_TimeInterval),        intent(in),  optional :: timeStep
type(ESMF_Alarm), dimension(:), intent(out), optional :: ringingAlarmList
integer,                        intent(out), optional :: ringingAlarmCount
integer,                        intent(out), optional :: rc
```

DESCRIPTION:

Advances the `clock`'s current time by one time step: either the `clock`'s, or the passed-in `timeStep` (see below). This method optionally returns a list and number of ringing `ESMF_Alarms`. See also method `ESMF_ClockGetRingingAlarms`.
The arguments are:

clock The object instance to advance.

[timeStep] Time step is performed with given timeStep, instead of the `ESMF_Clock`'s. Does not replace the `ESMF_Clock`'s timeStep; use `ESMF_ClockSet(clock, timeStep, ...)` for this purpose. Supports applications with variable time steps.

[ringingAlarmList] Returns the array of alarms that are ringing after the time step.

[ringingAlarmCount] The number of alarms ringing after the time step.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.4 ESMF_ClockCreate - Create a new ESMF Clock

INTERFACE:

```
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateNew(name, timeStep, startTime, stopTime, &
                             runDuration, runTimeStepCount, refTime, rc)
```

RETURN VALUE:

```
type(ESMF_Clock) :: ESMF_ClockCreateNew
```

ARGUMENTS:

```
character (len=*),       intent(in),  optional :: name
type(ESMF_TimeInterval), intent(in)            :: timeStep
type(ESMF_Time),         intent(in)            :: startTime
type(ESMF_Time),         intent(in),  optional :: stopTime
type(ESMF_TimeInterval), intent(in),  optional :: runDuration
integer,                 intent(in),  optional :: runTimeStepCount
type(ESMF_Time),         intent(in),  optional :: refTime
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Creates and sets the initial values in a new `ESMF_Clock`.
This is a private method; invoke via the public overloaded entry point `ESMF_ClockCreate()`.
The arguments are:

[name] The name for the newly created clock. If not specified, a default unique name will be generated: "ClockNNN" where NNN is a unique sequence number from 001 to 999.

timeStep The `ESMF_Clock`'s time step interval.

startTime The `ESMF_Clock`'s starting time.

[stopTime] The `ESMF_Clock`'s stopping time. If neither stopTime, runDuration, nor runTimeStepCount is specified, clock runs "forever"; user must use other means to know when to stop (e.g. ESMF_Alarm or ESMF_ClockGet(clock, currTime)). Mutually exclusive with runDuration and runTimeStepCount.

[runDuration] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + runDuration. Mutually exclusive with stopTime and runTimeStepCount.

[runTimeStepCount] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + (runTimeStepCount * timeStep). Mutually exclusive with stopTime and runDuration.

[refTime] The `ESMF_Clock`'s reference time. Provides reference point for simulation time (see currSimTime in ESMF_ClockGet() below).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.5 ESMF_ClockCreate - Create a copy of an existing ESMF Clock

INTERFACE:

```
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateCopy(clock, rc)
```

RETURN VALUE:

```
type(ESMF_Clock) :: ESMF_ClockCreateCopy
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in)            :: clock
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Creates a copy of a given `ESMF_Clock`.
This is a private method; invoke via the public overloaded entry point `ESMF_ClockCreate()`.
The arguments are:

clock The `ESMF_Clock` to copy.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.6 ESMF_ClockDestroy - Free all resources associated with a Clock

INTERFACE:

```
subroutine ESMF_ClockDestroy(clock, rc)
```

ARGUMENTS:

```
type(ESMF_Clock) :: clock
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this `ESMF_Clock`.
The arguments are:

clock Destroy contents of this `ESMF_Clock`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.7 ESMF_ClockGet - Get a Clock's properties

INTERFACE:

```
subroutine ESMF_ClockGet(clock, name, timeStep, startTime, stopTime, &
                         runDuration, runTimeStepCount, refTime, &
                         currTime, prevTime, currSimTime, prevSimTime, &
                         calendar, calendarType, timeZone, advanceCount, &
                         alarmCount, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),        intent(in)            :: clock
character (len=*),       intent(out), optional :: name
type(ESMF_TimeInterval), intent(out), optional :: timeStep
type(ESMF_Time),         intent(out), optional :: startTime
type(ESMF_Time),         intent(out), optional :: stopTime
type(ESMF_TimeInterval), intent(out), optional :: runDuration
real(ESMF_KIND_R8),      intent(out), optional :: runTimeStepCount
type(ESMF_Time),         intent(out), optional :: refTime
type(ESMF_Time),         intent(out), optional :: currTime
type(ESMF_Time),         intent(out), optional :: prevTime
type(ESMF_TimeInterval), intent(out), optional :: currSimTime
type(ESMF_TimeInterval), intent(out), optional :: prevSimTime
type(ESMF_Calendar),     intent(out), optional :: calendar
type(ESMF_CalendarType), intent(out), optional :: calendarType
integer,                 intent(out), optional :: timeZone
integer(ESMF_KIND_I8),   intent(out), optional :: advanceCount
integer,                 intent(out), optional :: alarmCount
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Gets one or more of the properties of an `ESMF_Clock`.
The arguments are:

clock The object instance to query.

[name] The name of this clock.

[timeStep] The `ESMF_Clock`'s time step interval.

[startTime] The `ESMF_Clock`'s starting time.

[stopTime] The `ESMF_Clock`'s stopping time.

[runDuration] Alternative way to get `ESMF_Clock`'s stopping time; runDuration = stopTime - startTime.

[runTimeStepCount] Alternative way to get `ESMF_Clock`'s stopping time; runTimeStepCount = (stopTime - startTime) / timeStep.

[refTime] The `ESMF_Clock`'s reference time.

[currTime] The `ESMF_Clock`'s current time.

[prevTime] The `ESMF_Clock`'s previous time. Equals currTime at the previous time step.

[currSimTime] The current simulation time (currTime - refTime).

[prevSimTime] The previous simulation time. Equals currSimTime at the previous time step.

[calendar] The `Calendar` on which all the `Clock`'s times are defined.

[calendarType] The `CalendarType` on which all the `Clock`'s times are defined.

[timeZone] The timezone within which all the `Clock`'s times are defined.

[advanceCount] The number of times the `ESMF_Clock` has been advanced.

[alarmCount] The number of `ESMF_Alarm`s in the `ESMF_Clock`'s `ESMF_Alarm` list.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.8 ESMF_ClockGetAlarm - Get an Alarm in a Clock's Alarm list

INTERFACE:

```
      subroutine ESMF_ClockGetAlarm(clock, name, alarm, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),  intent(in)            :: clock
      character (len=*), intent(in)            :: name
      type(ESMF_Alarm),  intent(out)           :: alarm
      integer,           intent(out), optional :: rc
```

DESCRIPTION:

Gets the `alarm` whose name is the value of name in the `clock`'s `ESMF_Alarm` list.
The arguments are:

clock The object instance to get the `ESMF_Alarm` from.

name The name of the desired `ESMF_Alarm`.

alarm The desired alarm.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

36.4.9 ESMF_ClockGetAlarmList - Get a list of Alarms from a Clock

INTERFACE:

```
      subroutine ESMF_ClockGetAlarmList(clock, alarmListType, &
                                        alarmList, alarmCount, timeStep, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),                intent(in)            :: clock
type(ESMF_AlarmListType),        intent(in)            :: alarmListType
type(ESMF_Alarm), dimension(:), intent(out)           :: alarmList
integer,                         intent(out)           :: alarmCount
type(ESMF_TimeInterval),         intent(in),  optional :: timeStep
integer,                         intent(out), optional :: rc
```

DESCRIPTION:

Gets the clock's list of alarms.
The arguments are:

clock The object instance from which to get an ESMF_Alarm list.

alarmListType The type of list to get: ESMF_ALARMLIST_ALL : Returns the ESMF_Clock's entire list of alarms.

ESMF_ALARMLIST_NEXTRINGING : Return only those alarms that will ring upon the next clock time step. Can optionally specify argument timeStep (see below) to use instead of the clock's. See also method ESMF_AlarmWillRingNext() for checking a single alarm.

ESMF_ALARMLIST_PREVRINGING: Return only those alarms that were ringing on the previous ESMF_Clock time step. See also method ESMF_AlarmWasPrevRinging() for checking a single alarm.

ESMF_ALARMLIST_RINGING : Returns only those clock alarms that are currently ringing. See also method ESMF_ClockAdvance() for getting the list of ringing alarms subsequent to a time step. See also method ESMF_AlarmIsRinging() for checking a single alarm.

alarmList The array of returned alarms.

alarmCount The number of ESMF_Alarms in the returned list.

[timeStep] Optional time step to be used instead of the clock's. Only used with ESMF_ALARMLIST_NEXTRINGING alarmListType (see above); ignored if specified with other alarmListTypes.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.10 ESMF_ClockGetNextTime - Calculate a Clock's next time

INTERFACE:

```
subroutine ESMF_ClockGetNextTime(clock, nextTime, timeStep, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),        intent(in)            :: clock
type(ESMF_Time),         intent(out)           :: nextTime
type(ESMF_TimeInterval), intent(in),  optional :: timeStep
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Calculates what the next time of the clock will be, based on the clock's current time step or an optionally passed-in timeStep.
The arguments are:

clock The object instance for which to get the next time.

nextTime The resulting ESMF_Clock's next time.

[timeStep] The time step interval to use instead of the clock's.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.11 ESMF_ClockIsStopTime - Test if the Clock has reached or exceeded its stop time

INTERFACE:

```
function ESMF_ClockIsStopTime(clock, rc)
```

RETURN VALUE:

```
logical :: ESMF_ClockIsStopTime
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in)            :: clock
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Returns true if the clock has reached or exceeded its stop time, and false otherwise.
The arguments are:

clock The object instance to check.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.12 ESMF_ClockPrint - Print the contents of a Clock

INTERFACE:

```
subroutine ESMF_ClockPrint(clock, options, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),  intent(in)            :: clock
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Prints out an ESMF_Clock's properties to stdout, in support of testing and debugging. The options control the type of information and level of detail.
The arguments are:

clock ESMF_Clock to be printed out.

[options] Print options. If none specified, prints all clock property values.
"advanceCount" - print the number of times the clock has been advanced.
"alarmCount" - print the number of alarms in the clock's list.
"alarmList" - print the clock's alarm list.
"currTime" - print the current clock time.
"name" - print the clock's name.
"prevTime" - print the previous clock time.
"refTime" - print the clock's reference time.
"startTime" - print the clock's start time.
"stopTime" - print the clock's stop time.
"timeStep" - print the clock's time step.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.13 ESMF_ClockSet - Set one or more properties of a Clock

INTERFACE:

```
subroutine ESMF_ClockSet(clock, name, timeStep, startTime, stopTime, &
                         runDuration, runTimeStepCount, refTime, &
                         currTime, advanceCount, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),         intent(inout)          :: clock
character (len=*),        intent(in),  optional :: name
type(ESMF_TimeInterval), intent(in),  optional :: timeStep
type(ESMF_Time),          intent(in),  optional :: startTime
type(ESMF_Time),          intent(in),  optional :: stopTime
type(ESMF_TimeInterval), intent(in),  optional :: runDuration
integer,                  intent(in),  optional :: runTimeStepCount
type(ESMF_Time),          intent(in),  optional :: refTime
type(ESMF_Time),          intent(in),  optional :: currTime
integer(ESMF_KIND_I8),    intent(in),  optional :: advanceCount
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Sets/resets one or more of the properties of an ESMF_Clock that was previously initialized via ESMF_ClockCreate(). The arguments are:

clock The object instance to set.

[name] The new name for this clock.

[timeStep] The ESMF_Clock's time step interval. This is used to change a clock's timestep property for those applications that need variable timesteps. Also see ESMF_ClockAdvance() below for specifying variable timesteps that are NOT saved as the clock's internal time step property.

[startTime] The ESMF_Clock's starting time.

[stopTime] The ESMF_Clock's stopping time. If neither stopTime, runDuration, nor runTimeStepCount is specified, clock runs "forever"; user must use other means to know when to stop (e.g. ESMF_Alarm or ESMF_ClockGet(clock, currTime)). Mutually exclusive with runDuration and runTimeStepCount.

[runDuration] Alternative way to specify ESMF_Clock's stopping time; stopTime = startTime + runDuration. Mutually exclusive with stopTime and runTimeStepCount.

[runTimeStepCount] Alternative way to specify ESMF_Clock's stopping time; stopTime = startTime + (runTimeStepCount * timeStep). Mutually exclusive with stopTime and runDuration.

[refTime] The ESMF_Clock's reference time. See description in ESMF_ClockCreate() above.

[currTime] The current time.

[advanceCount] The number of times the clock has been timestepped.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.14 ESMF_ClockSyncToRealTime - Set Clock's current time to wall clock time

INTERFACE:

```
subroutine ESMF_ClockSyncToRealTime(clock, rc)
```

ARGUMENTS:

```
type(ESMF_Clock), intent(inout)          :: clock
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Sets a clock's current time to the wall clock time. It is accurate to the nearest second.
The arguments are:

clock The object instance to be synchronized with wall clock time.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.4.15 ESMF_ClockValidate - Validate a Clock's properties

INTERFACE:

```
subroutine ESMF_ClockValidate(clock, options, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),  intent(in)            :: clock
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Checks whether a clock is valid.
The arguments are:

clock ESMF_Clock to be validated.

[options] Validation options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

37 Alarm Class

37.1 Description

The Alarm class identifies events that occur at specific Times or specific TimeIntervals by returning a true value at those times or subsequent times, and a false value otherwise.

37.2 Alarm Options

37.2.1 ESMF_AlarmListType

DESCRIPTION:
Specifies the characteristics of Alarms that populate a retrieved Alarm list.
Valid values are:

ESMF_ALARMLIST_ALL All alarms.

ESMF_ALARMLIST_NEXTRINGING Alarms that will ring before or at the next timestep.

ESMF_ALARMLIST_PREVRINGING Alarms that rang at or since the last timestep.

ESMF_ALARMLIST_RINGING Only ringing alarms.

37.3 Use and Examples

Alarms are used in conjunction with Clocks (see Section 36.1). Multiple Alarms can be associated with a Clock. During the `ESMF_ClockAdvance` method, a Clock iterates over its internal Alarms to determine if any are ringing. Alarms ring when a specified Alarm time is reached or exceeded, taking into account whether the time step is positive or negative. On completion of the time advance call, the Clock optionally returns a list of ringing alarms.
Each ringing Alarm can then be processed using Alarm methods for identifying, turning off, disabling or resetting the Alarm.
Alarm methods are defined for obtaining the ringing state, turning the ringer on/off, enabling/disabling the Alarm, and getting/setting associated times.
The following example shows how to set and process Alarms.

```
! !PROGRAM: ESMF_AlarmEx - Alarm examples
!
! !DESCRIPTION:
!
! This program shows an example of how to create, initialize, and process
! alarms associated with a clock.
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF_Mod
      implicit none

      ! instantiate time_step, start, stop, and alarm times
      type(ESMF_TimeInterval) :: timeStep, alarmInterval
      type(ESMF_Time) :: alarmTime, startTime, stopTime

      ! instantiate a clock
      type(ESMF_Clock) :: clock

      ! instantiate Alarm lists
      integer, parameter :: NUMALARMS = 2
      type(ESMF_Alarm) :: alarm(NUMALARMS)
```

```
! local variables for Get methods
integer :: ringingAlarmCount  ! at any time step (0 to NUMALARMS)

! name, loop counter, result code
character (len=ESMF_MAXSTR) :: name
integer :: i, rc


! initialize ESMF framework
call ESMF_Initialize(defaultCalendar=ESMF_CAL_GREGORIAN, rc=rc)
```

37.3.1 Clock Initialization

This example shows how to create and initialize an `ESMF_Clock`.

```
! initialize time interval to 1 day
call ESMF_TimeIntervalSet(timeStep, d=1, rc=rc)


! initialize start time to 9/1/2003
call ESMF_TimeSet(startTime, yy=2003, mm=9, dd=1, rc=rc)


! initialize stop time to 9/30/2003
call ESMF_TimeSet(stopTime, yy=2003, mm=9, dd=30, rc=rc)


! create & initialize the clock with the above values
clock = ESMF_ClockCreate("The Clock", timeStep, startTime, stopTime, &
                         rc=rc)
```

37.3.2 Alarm Initialization

This example shows how to create and initialize two `ESMF_Alarms` and associate them with the clock.

```
! Initialize first alarm to be a one-shot on 9/15/2003 and associate
! it with the clock
call ESMF_TimeSet(alarmTime, yy=2003, mm=9, dd=15, rc=rc)


alarm(1) = ESMF_AlarmCreate("Example alarm 1", clock, &
                            ringTime=alarmTime, rc=rc)


! Initialize second alarm to ring on a 1 week interval starting 9/1/2003
! and associate it with the clock
call ESMF_TimeSet(alarmTime, yy=2003, mm=9, dd=1, rc=rc)


call ESMF_TimeIntervalSet(alarmInterval, d=7, rc=rc)


! Alarm gets default name "Alarm002"
alarm(2) = ESMF_AlarmCreate(clock=clock, ringTime=alarmTime, &
                            ringInterval=alarmInterval, rc=rc)
```

37.3.3 Clock Advance and Alarm Processing

This example shows how to advance an ESMF_Clock and process any resulting ringing alarms.

```
! time step clock from start time to stop time
do while (.not.ESMF_ClockIsStopTime(clock, rc))


  ! perform time step and get the number of any ringing alarms
  call ESMF_ClockAdvance(clock, ringingAlarmCount=ringingAlarmCount, &
                         rc=rc)


  call ESMF_ClockPrint(clock, "currTime string", rc)


  ! check if alarms are ringing
  if (ringingAlarmCount > 0) then
    print *, "number of ringing alarms = ", ringingAlarmCount

    do i = 1, NUMALARMS
      if (ESMF_AlarmIsRinging(alarm(i), rc)) then


        call ESMF_AlarmGet(alarm(i), name=name, rc=rc)
        print *, trim(name), " is ringing!"


        ! after processing alarm, turn it off
        call ESMF_AlarmRingerOff(alarm(i), rc)


      end if ! this alarm is ringing
    end do ! each ringing alarm
  endif ! ringing alarms
end do ! timestep clock
```

37.3.4 Alarm and Clock Destruction

This example shows how to destroy ESMF_Alarms and ESMF_Clocks.

```
call ESMF_AlarmDestroy(alarm(1), rc=rc)


call ESMF_AlarmDestroy(alarm(2), rc=rc)


call ESMF_ClockDestroy(clock, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc)


end program ESMF_AlarmEx
```

37.4 Restrictions and Future Work

1. **Limit on number of Alarms.** The maximum number of Alarms that can be associated with a Clock is currently hard-coded at 100. This is done in both Fortran and C++ with a #define for ease of modification.

37.5 Design and Implementation Notes

The Alarm class is designed as a deep, dynamically allocatable class, based on a pointer type. This allows for both indirect and direct manipulation of alarms. Indirect alarm manipulation is where ESMF_Alarm API methods, such as ESMF_AlarmRingerOff(), are invoked on alarm references (pointers) returned from ESMF_Clock queries such as "return ringing alarms." Since the method is performed on an alarm reference, the actual alarm held by the clock is affected, not just a user's local copy. Direct alarm manipulation is the more common case where alarm API methods are invoked on the original alarm objects created by the user.
For consistency, the ESMF_Clock class is also designed as a deep, dynamically allocatable class.
An additional benefit from this approach is that Clocks and Alarms can be created and used from anywhere in a user's code without regard to the scope in which they were created. In constrast, statically created Alarms and Clocks would disappear if created within a user's routine that returns, whereas dynamically allocated Alarms and Clocks will persist until explicitly destroyed by the user.

37.6 Class API

37.6.1 ESMF_AlarmOperator(==) - Test if Alarm 1 is equal to Alarm 2

INTERFACE:

```
interface operator(==)
if (alarm1 == alarm2) then ... endif
             OR
result = (alarm1 == alarm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in) :: alarm1
type(ESMF_Alarm), intent(in) :: alarm2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Alarm` class. Compare two alarms for equality; return true if equal, false otherwise. Comparison is based on IDs, which are distinct for newly created alarms and identical for alarms created as copies.
The arguments are:

alarm1 The first `ESMF_Alarm` in comparison.

alarm2 The second `ESMF_Alarm` in comparison.

37.6.2 ESMF_AlarmOperator(/=) - Test if Alarm 1 is not equal to Alarm 2

INTERFACE:

```
interface operator(/=)
if (alarm1 /= alarm2) then ... endif
             OR
result = (alarm1 /= alarm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in) :: alarm1
type(ESMF_Alarm), intent(in) :: alarm2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Alarm` class. Compare two alarms for inequality; return true if not equal, false otherwise. Comparison is based on IDs, which are distinct for newly created alarms and identical for alarms created as copies.
The arguments are:

alarm1 The first `ESMF_Alarm` in comparison.

alarm2 The second `ESMF_Alarm` in comparison.

37.6.3 ESMF_AlarmCreate - Create a new ESMF Alarm

INTERFACE:

```
! Private name; call using ESMF_AlarmCreate()
function ESMF_AlarmCreateNew(name, clock, ringTime, ringInterval, &
                             stopTime, ringDuration, &
                             ringTimeStepCount, &
                             refTime, enabled, sticky, rc)
```

RETURN VALUE:

```
type(ESMF_Alarm) :: ESMF_AlarmCreateNew
```

ARGUMENTS:

```
character (len=*),       intent(in),  optional :: name
type(ESMF_Clock),        intent(in)            :: clock
type(ESMF_Time),         intent(in),  optional :: ringTime
type(ESMF_TimeInterval), intent(in),  optional :: ringInterval
type(ESMF_Time),         intent(in),  optional :: stopTime
type(ESMF_TimeInterval), intent(in),  optional :: ringDuration
integer,                 intent(in),  optional :: ringTimeStepCount
type(ESMF_Time),         intent(in),  optional :: refTime
logical,                 intent(in),  optional :: enabled
logical,                 intent(in),  optional :: sticky
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Creates and sets the initial values in a new `ESMF_Alarm`.
This is a private method; invoke via the public overloaded entry point `ESMF_AlarmCreate()`.
The arguments are:

[name] The name for the newly created alarm. If not specified, a default unique name will be generated: "AlarmNNN" where NNN is a unique sequence number from 001 to 999.

clock The clock with which to associate this newly created alarm.

[ringTime] The ring time for a one-shot alarm or the first ring time for a repeating (interval) alarm. Must specify at least one of ringTime or ringInterval.

[ringInterval] The ring interval for repeating (interval) alarms. If `ringTime` is not also specified (first ring time), it will be calculated as the `clock`'s current time plus `ringInterval`. Must specify at least one of ringTime or ringInterval.

[stopTime] The stop time for repeating (interval) alarms. If not specified, an interval alarm will repeat forever.

[ringDuration] The absolute ring duration. If not sticky (see argument below), alarms rings for ringDuration, then turns itself off. Mutually exclusive with ringTimeStepCount (below); used only if ringTimeStepCount is zero. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[ringTimeStepCount] The relative ring duration. If not sticky (see argument below), alarms rings for ringTimeStepCount, then turns itself off. Mutually exclusive with ringDuration (above); used if non-zero, otherwise ringDuration is used. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[refTime] The reference (i.e. base) time for an interval alarm.

[enabled] Sets the enabled state; default is on (true). If disabled, an alarm will not function at all. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] Sets the sticky state; default is on (true). If sticky, once an alarm is ringing, it will remain ringing until turned off manually via a user call to `ESMF_AlarmRingerOff()`. If not sticky, an alarm will turn itself off after a certain ring duration specified by either ringDuration or ringTimeStepCount (see above). See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.4 ESMF_AlarmCreate - Create a copy of an existing ESMF Alarm

INTERFACE:

```
! Private name; call using ESMF_AlarmCreate()
function ESMF_AlarmCreateCopy(alarm, rc)
```

RETURN VALUE:

```
type(ESMF_Alarm) :: ESMF_AlarmCreateCopy
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in)            :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Creates a copy of a given ESMF_Alarm.
This is a private method; invoke via the public overloaded entry point ESMF_AlarmCreate().
The arguments are:

alarm The ESMF_Alarm to copy.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

37.6.5 ESMF_AlarmDestroy - Free all resources associated with an Alarm

INTERFACE:

```
subroutine ESMF_AlarmDestroy(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm) :: alarm
integer, intent(out), optional :: rc
```

DESCRIPTION:

Releases all resources associated with this ESMF_Alarm.
The arguments are:

alarm Destroy contents of this ESMF_Alarm.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

37.6.6 ESMF_AlarmDisable - Disable an Alarm

INTERFACE:

```
subroutine ESMF_AlarmDisable(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(inout)          :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Disables an ESMF_Alarm.
The arguments are:

alarm The object instance to disable.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

37.6.7 ESMF_AlarmEnable - Enable an Alarm

INTERFACE:

```
subroutine ESMF_AlarmEnable(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(inout)          :: alarm
integer,          intent(out), optional  :: rc
```

DESCRIPTION:

Enables an `ESMF_Alarm` to function.
The arguments are:

alarm The object instance to enable.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.8 ESMF_AlarmGet - Get Alarm properties

INTERFACE:

```
subroutine ESMF_AlarmGet(alarm, name, clock, ringTime, prevRingTime, &
                         ringInterval, stopTime, ringDuration, &
                         ringTimeStepCount, timeStepRingingCount, &
                         ringBegin, refTime, ringing, &
                         ringingOnPrevTimeStep, enabled, sticky, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm),        intent(in)            :: alarm
character (len=*),       intent(out), optional :: name
type(ESMF_Clock),        intent(out), optional :: clock
type(ESMF_Time),         intent(out), optional :: ringTime
type(ESMF_Time),         intent(out), optional :: prevRingTime
type(ESMF_TimeInterval), intent(out), optional :: ringInterval
type(ESMF_Time),         intent(out), optional :: stopTime
type(ESMF_TimeInterval), intent(out), optional :: ringDuration
integer,                 intent(out), optional :: ringTimeStepCount
integer,                 intent(out), optional :: timeStepRingingCount
type(ESMF_Time),         intent(out), optional :: ringBegin
type(ESMF_Time),         intent(out), optional :: refTime
logical,                 intent(out), optional :: ringing
logical,                 intent(out), optional :: ringingOnPrevTimeStep
logical,                 intent(out), optional :: enabled
logical,                 intent(out), optional :: sticky
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Gets one or more of an `ESMF_Alarm`'s properties.
The arguments are:

alarm The object instance to query.

[name] The name of this alarm.

[clock] The associated clock.

[ringTime] The ring time for a one-shot alarm or the next repeating alarm.

[prevRingTime] The previous ring time.

[ringInterval] The ring interval for repeating (interval) alarms.

[stopTime] The stop time for repeating (interval) alarms.

[ringDuration] The ring duration. Mutually exclusive with ringTimeStepCount (see below).

[ringTimeStepCount] The number of time steps comprising the ring duration. Mutually exclusive with ringDuration (see above).

[timeStepRingingCount] The number of time steps for which the alarm has been ringing thus far. Used internally for tracking ringTimeStepCount ring durations (see above). Mutually exclusive with ringBegin (see below).

[ringBegin] The time when the alarm began ringing. Used internally for tracking ringDuration (see above). Mutually exclusive with timeStepRingingCount (see above).

[refTime] The reference (i.e. base) time for an interval alarm.

[ringing] The current ringing state. See also `ESMF_AlarmRingerOn()`, `ESMF_AlarmRingerOff()`.

[ringingOnPrevTimeStep] The ringing state upon the previous time step. Same as `ESMF_AlarmWasPrevRinging()`.

[enabled] The enabled state. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] The sticky state. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

37.6.9 ESMF_AlarmIsEnabled - Check if Alarm is enabled

INTERFACE:

```
function ESMF_AlarmIsEnabled(alarm, rc)
```

RETURN VALUE:

```
logical :: ESMF_AlarmIsEnabled
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in)            :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Check if `ESMF_Alarm` is enabled.
The arguments are:

alarm The object instance to check for enabled state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.10 ESMF_AlarmIsRinging - Check if Alarm is ringing

INTERFACE:

```
function ESMF_AlarmIsRinging(alarm, rc)
```

RETURN VALUE:

```
logical :: ESMF_AlarmIsRinging
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in)            :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Check if `ESMF_Alarm` is ringing.
See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_RINGING, ...)` to get a list of all ringing alarms belonging to an `ESMF_Clock`.
The arguments are:

alarm The alarm to check for ringing state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.11 ESMF_AlarmIsSticky - Check if Alarm is sticky

INTERFACE:

```
function ESMF_AlarmIsSticky(alarm, rc)
```

RETURN VALUE:

```
logical :: ESMF_AlarmIsSticky
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in)            :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Check if `alarm` is sticky.
The arguments are:

alarm The object instance to check for sticky state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.12 ESMF_AlarmNotSticky - Unset an Alarm's sticky flag

INTERFACE:

```
subroutine ESMF_AlarmNotSticky(alarm, ringDuration, &
                               ringTimeStepCount, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm),        intent(inout)          :: alarm
type(ESMF_TimeInterval), intent(in),  optional :: ringDuration
integer,                 intent(in),  optional :: ringTimeStepCount
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Unset an `ESMF_Alarm`'s sticky flag; once alarm is ringing, it turns itself off after ringDuration.
The arguments are:

alarm The object instance to unset sticky.

[ringDuration] If not sticky, alarms rings for ringDuration, then turns itself off.

[ringTimeStepCount] If not sticky, alarms rings for ringTimeStepCount, then turns itself off.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.13 ESMF_AlarmPrint - Print out an Alarm's properties

INTERFACE:

```
subroutine ESMF_AlarmPrint(alarm, options, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm),   intent(in)             :: alarm
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Prints out an `ESMF_Alarm`'s properties to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.
The arguments are:

alarm `ESMF_Alarm` to be printed out.

[options] Print options. If none specified, prints all `alarm` property values.
"clock" - print the associated clock's name.
"enabled" - print the alarm's ability to ring.
"name" - print the alarm's name.
"prevRingTime" - print the alarm's previous ring time.
"ringBegin" - print time when the alarm actually begins to ring.

"ringDuration" - print how long this alarm is to remain ringing.
"ringing" - print the alarm's current ringing state.
"ringingOnPrevTimeStep" - print whether the alarm was ringing immediately after the previous clock time step.
"ringInterval" - print the alarm's periodic ring interval.
"ringTime" - print the alarm's next time to ring.
"ringTimeStepCount" - print how long this alarm is to remain ringing, in terms of a number of clock time steps.
"refTime" - print the alarm's interval reference (base) time.
"sticky" - print whether the alarm must be turned off manually.
"stopTime" - print when alarm intervals end.
"timeStepRingingCount" - print the number of time steps the alarm has been ringing thus far.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.14 ESMF_AlarmRingerOff - Turn off an Alarm

INTERFACE:

```
subroutine ESMF_AlarmRingerOff(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(inout)          :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Turn off an `ESMF_Alarm`; unsets ringing state.
The arguments are:

alarm The object instance to turn off.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.15 ESMF_AlarmRingerOn - Turn on an Alarm

INTERFACE:

```
subroutine ESMF_AlarmRingerOn(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(inout)          :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Turn on an `ESMF_Alarm`; sets ringing state.
The arguments are:

alarm The object instance to turn on.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.16 ESMF_AlarmSet - Set Alarm properties

INTERFACE:

```
subroutine ESMF_AlarmSet(alarm, name, clock, ringTime, ringInterval, &
                         stopTime, ringDuration, ringTimeStepCount, &
                         refTime, ringing, enabled, sticky, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm),         intent(inout)          :: alarm
character (len=*),        intent(in),  optional :: name
type(ESMF_Clock),         intent(in),  optional :: clock
type(ESMF_Time),          intent(in),  optional :: ringTime
type(ESMF_TimeInterval), intent(in),  optional :: ringInterval
type(ESMF_Time),          intent(in),  optional :: stopTime
type(ESMF_TimeInterval), intent(in),  optional :: ringDuration
integer,                  intent(in),  optional :: ringTimeStepCount
type(ESMF_Time),          intent(in),  optional :: refTime
logical,                  intent(in),  optional :: ringing
logical,                  intent(in),  optional :: enabled
logical,                  intent(in),  optional :: sticky
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Sets/resets one or more of the properties of an `ESMF_Alarm` that was previously initialized via `ESMF_AlarmCreate()`. The arguments are:

alarm The object instance to set.

[name] The new name for this alarm.

[clock] Re-associates this alarm with a different clock.

[ringTime] The next ring time for a one-shot alarm or a repeating (interval) alarm.

[ringInterval] The ring interval for repeating (interval) alarms.

[stopTime] The stop time for repeating (interval) alarms.

[ringDuration] The absolute ring duration. If not sticky (see argument below), alarms rings for ringDuration, then turns itself off. Mutually exclusive with ringTimeStepCount (below); used only if ringTimeStepCount is zero. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[ringTimeStepCount] The relative ring duration. If not sticky (see argument below), alarms rings for ringTimeStepCount, then turns itself off. Mutually exclusive with ringDuration (above); used if non-zero, otherwise ringDuration is used. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[refTime] The reference (i.e. base) time for an interval alarm.

[ringing] Sets the ringing state. See also `ESMF_AlarmRingerOn()`, `ESMF_AlarmRingerOff()`.

[enabled] Sets the enabled state. If disabled, an alarm will not function at all. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] Sets the sticky state. If sticky, once an alarm is ringing, it will remain ringing until turned off manually via a user call to `ESMF_AlarmRingerOff()`. If not sticky, an alarm will turn itself off after a certain ring duration specified by either ringDuration or ringTimeStepCount (see above). See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.17 ESMF_AlarmSticky - Set an Alarm's sticky flag

INTERFACE:

```
subroutine ESMF_AlarmSticky(alarm, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(inout)          :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an `ESMF_Alarm`'s sticky flag; once alarm is ringing, it remains ringing until `ESMF_AlarmRingerOff()` is called.
The arguments are:

alarm The object instance to be set sticky.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.18 ESMF_AlarmValidate - Validate an Alarm's properties

INTERFACE:

```
subroutine ESMF_AlarmValidate(alarm, options, rc)
```

ARGUMENTS:

```
type(ESMF_Alarm),  intent(in)            :: alarm
character (len=*), intent(in),  optional :: options
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Performs a validation check on an `ESMF_Alarm`'s properties.
The arguments are:

alarm `ESMF_Alarm` to be validated.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.19 ESMF_AlarmWasPrevRinging - Check if Alarm was ringing on the previous Clock timestep

INTERFACE:

```
function ESMF_AlarmWasPrevRinging(alarm, rc)
```

RETURN VALUE:

```
logical :: ESMF_AlarmWasPrevRinging
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in)            :: alarm
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Check if `ESMF_Alarm` was ringing on the previous clock timestep.
See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_PREVRINGING, ...)` get a list of all alarms belonging to a `ESMF_Clock` that were ringing on the previous time step.
The arguments are:

alarm The object instance to check for previous ringing state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

37.6.20 ESMF_AlarmWillRingNext - Check if Alarm will ring upon the next Clock timestep

INTERFACE:

```
function ESMF_AlarmWillRingNext(alarm, timeStep, rc)
```

RETURN VALUE:

```
logical :: ESMF_AlarmWillRingNext
```

ARGUMENTS:

```
type(ESMF_Alarm),        intent(in)            :: alarm
type(ESMF_TimeInterval), intent(in),  optional :: timeStep
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Check if `ESMF_Alarm` will ring on the next clock timestep, either the current clock timestep or a passed-in timestep.
See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_NEXTRINGING, ...)` to get a list of all alarms belonging to a `ESMF_Clock` that will ring on the next time step.
The arguments are:

alarm The alarm to check for next ringing state.

[timeStep] Optional timestep to use instead of the clock's.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38 Config Class

38.1 Description

ESMF Configuration Management is based on NASA DAO's Inpak 90 package, a Fortran 90 collection of routines/functions for accessing *Resource Files* in ASCII format.The package is optimized for minimizing formatted I/O, performing all of its string operations in memory using Fortran intrinsic functions.

Module `ESMF_ConfigMod` is implemented in Fortran.

38.2 Use and Examples

38.2.1 Resource Files

A *Resource File* is a text file consisting of variable length lines (records), each possibly starting with a *label* (or *key*), followed by some data. A simple resource file looks like this:

```
# Lines starting with # are comments which are
# ignored during processing.
my_file_names:          jan87.dat jan88.dat jan89.dat
radius_of_the_earth:    6.37E6  # these are comments too
constants:              3.1415   25
my_favourite_colors:    green blue 022 # text & number are OK
```

In this example, `my_file_names:` and `constants:` are labels, while `jan87.dat, jan88.dat` and `jan89.dat` are data associated with label `my_file_names:`. Resource files can also contain simple tables of the form,

```
my_table_name::
 1000     3000     263.0
  925     3000     263.0
  850     3000     263.0
  700     3000     269.0
  500     3000     287.0
  400     3000     295.8
  300     3000     295.8
::
```

Resource files are intended for random access (except between ::'s in a table definition). Normally, the order of records should not be important. However, the order of records may be important if the same label appears multiple times.

38.2.2 A Quick Overview

The first step is to create the ESMF Config and load the ASCII resource (rc) file into memory[5]:

```
      cf = ESMF_ConfigCreate (layout, rc)
      call ESMF_ConfigLoadFile (cf, fname, rc = rc)
```

The next step is to select the label (record) of interest, say

```
      call ESMF_ConfigFindLabel  ( cf, 'constants:', rc = rc )
```

The 2 constants above can be retrieved with the following code fragment:

[5] See next section for a complete description of parameters for each routine/function

```
real    r
integer i
call ESMF_ConfigFindLabel( cf, 'constants:', rc = rc)
r = ESMF_ConfigGetFloat( cf, rc = rc )        results in r = 3.1415
i = ESMF_ConfigGetInt( cf, rc = rc )          results in i = 25
```

The file names above can be retrieved with the following code fragment:

```
character*20 fn1, fn2, fn3
integer      rc
call ESMF_ConfigFindLabel ( cf, 'my_file_names:', rc = rc )
call ESMF_ConfigGetString ( cf, fn1, rc = rc )   ==> fn1 = 'jan87.dat'
call ESMF_ConfigGetString ( cf, fn2, rc = rc )   ==> fn1 = 'jan88.dat'
call ESMF_ConfigGetString ( cf, fn3, rc = rc )   ==> fn1 = 'jan89.dat'
```

To access the table above, the user first must use **ESMF_ConfigFindLabel**() to locate the beginning of the table, e.g.,

```
call ESMF_ConfigFindLabel(cf, 'my_table_name::', rc = rc)
```

Subsequently, **callESMF_ConfigNextLine**() can be used to gain access to each row of the table. Here is a code fragment to read the above table (7 rows, 3 columns):

```
real          table(7,3)
character*20  word
integer       rc
call  ESMF_ConfigFindLabel(cf, 'my_table_name::', rc = rc)
do i = 1, 7
   call  call ESMF_ConfigNextLine( cf, rc = rc )
   do j = 1, 3
      table(i,j) = ESMF_ConfigGetFloat( cf, rc = rc )
   end do
end do
```

The work with the configuration `cf` is finalized by call to **ESMF_ConfigDestroy**():

```
integer rc
call ESMF_ConfigDestroy( cf, rc )
```

Common Arguments:

```
character*(*)      fname       file name
integer            rc        error return code (0 is OK)
character*(*)      label       label (key) to locate record
character*(*)      word        blank delimited string
character*(*)      string      a sequence of characters
```

38.2.3 Package History

The ESMF Configuration Management Package was evolved by Leonid Zaslavsky and Arlindo da Silva from Ipack90 package created by Arlindo da Silva at NASA DAO.
Back in the 70's Eli Isaacson wrote IOPACK in Fortran 66. In June of 1987 Arlindo da Silva wrote Inpak77 using Fortran 77 string functions; Inpak 77 is a vastly simplified IOPACK, but has its own goodies not found in IOPACK. Inpak 90 removes some obsolete functionality in Inpak77, and parses the whole resource file in memory for performance.

38.3 Class API

38.3.1 ESMF_ConfigCreate - Create a Config object

INTERFACE:

```
type(ESMF_Config) function ESMF_ConfigCreate( rc )
```

ARGUMENTS:

```
integer,intent(out), optional              :: rc
```

DESCRIPTION:

Creates an `ESMF_Config` for use in subsequent calls.
The arguments are:

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.2 ESMF_ConfigDestroy - Destroy a Config object

INTERFACE:

```
subroutine ESMF_ConfigDestroy( config, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout) :: config
integer,intent(out), optional    :: rc
```

DESCRIPTION:

Destroys the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.3 ESMF_ConfigFindLabel - Find a label

INTERFACE:

```
subroutine ESMF_ConfigFindLabel( config, label, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)  :: config
character(len=*), intent(in)     :: label
integer, intent(out), optional   :: rc
```

DESCRIPTION:

Finds the `label` (key) in the `config` file.
Since the search is done by looking for a word in the whole resource file, it is important to use special conventions to distinguish labels from other words in the resource files. The DAO convention is to finish line labels by : and table labels by ::.
The arguments are:

config Already created `ESMF_Config` object.

label Identifying label.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors. Equals -1 if buffer could not be loaded, -2 if label not found, and -3 if invalid operation with index.

38.3.4 ESMF_ConfigGetAttribute - Get a character string

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetString( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
character(len=*), intent(out)          :: value
character(len=*), intent(in), optional :: label
character(len=*), intent(in), optional :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets a sequence of characters. It will be terminated by the first white space.
The arguments are:

config Already created `ESMF_Config` object.

value Returned value.

[label] Identifing label.

[default] Default value if `label` is not found in `config` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.5 ESMF_ConfigGetAttribute - Get a 4-byte real number

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetFloatR4( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
real(ESMF_KIND_R4), intent(out)        :: value
character(len=*), intent(in), optional :: label
real, intent(in), optional             :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets a 4-byte real `value` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

value Returned value.

[label] Identifying label.

[default] Default value if `label` is not found in `config` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.6 ESMF_ConfigGetAttribute - Get an 8-byte real number

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetFloatR8( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
real(ESMF_KIND_R8), intent(out)        :: value
character(len=*), intent(in), optional :: label
real, intent(in), optional             :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets an 8-byte real `value` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

value Returned real value.

[label] Identifying label.

[default] Default value if `label` is not found in `config` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.7 ESMF_ConfigGetAttribute - Get a list of 4-byte real numbers

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetFloatsR4( config, valueList, count, label,  &
                                   default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
real(ESMF_KIND_R4), intent(inout)      :: valueList(:)
integer, intent(in)                    :: count
character(len=*), intent(in), optional :: label
real, intent(in), optional             :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets a 4-byte real `valueList` of a given `count` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

valueList Returned real values.

count Number of returned values expected.

[label] Identifing label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.8 ESMF_ConfigGetAttribute - Get a list of 8-byte real numbers

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetFloatsR8( config, valueList, count, label,  &
                                   default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
real(ESMF_KIND_R8), intent(inout)      :: valueList(:)
integer, intent(in)                    :: count
character(len=*), intent(in), optional :: label
real, intent(in), optional             :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets an 8-byte real `valueList` of a given `count` from the `config` object.
The arguments are:

config Already created ESMF_Config object.

valueList Returned values.

count Number of returned values expected.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

38.3.9 ESMF_ConfigGetAttribute - Get a 4-byte integer number

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetIntI4( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
integer(ESMF_KIND_I4), intent(out)     :: value
character(len=*), intent(in), optional :: label
integer, intent(in), optional          :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets an integer value from the config object.
The arguments are:

config Already created ESMF_Config object.

value Returned integer value.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

38.3.10 ESMF_ConfigGetAttribute - Get an 8-byte integer number

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetIntI8( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
integer(ESMF_KIND_I8), intent(out)     :: value
character(len=*), intent(in), optional :: label
integer, intent(in), optional          :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets an 8-byte integer `value` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

value Returned integer value.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.11 ESMF_ConfigGetAttribute - Get a list of 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetIntsI4( config, valueList, count, label,  &
                                 default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)       :: config
integer(ESMF_KIND_I4), intent(inout)   :: valueList(:)
integer, intent(in)                    :: count
character(len=*), intent(in), optional :: label
integer, intent(in), optional          :: default
integer, intent(out), optional         :: rc
```

DESCRIPTION:

Gets a 4-byte integer `valueList` of given `count` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

valueList Returned values.

count Number of returned values expected.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.12 ESMF_ConfigGetAttribute - Get a list of 8-byte integers

INTERFACE:

```
! Private name; call using ESMF_ConfigGetAttribute()
subroutine ESMF_ConfigGetIntsI8( config, valueList, count, label,  &
                                 default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)        :: config
integer(ESMF_KIND_I8), intent(inout)    :: valueList(:)
integer, intent(in)                     :: count
character(len=*), intent(in), optional :: label
integer, intent(in), optional           :: default
integer, intent(out), optional          :: rc
```

DESCRIPTION:

Gets an 8-byte integer `valueList` of given `count` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

valueList Returned values.

count Number of returned values expected.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.13 ESMF_ConfigGetChar - Get a character

INTERFACE:

```
subroutine ESMF_ConfigGetChar( config, value, label, default, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout)        :: config
character, intent(out)                  :: value
character(len=*), intent(in), optional :: label
character, intent(in), optional         :: default
integer, intent(out), optional          :: rc
```

DESCRIPTION:

Gets a character `value` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

value Returned value.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.14 ESMF_ConfigGetDim - Get table sizes

INTERFACE:

```
subroutine ESMF_ConfigGetDim( config, label, lineCount, columnCount, rc )

  implicit none

  type(ESMF_Config), intent(inout)       :: config    ! ESMF Configuration

 integer, intent(out)                    :: lineCount
 integer, intent(out)                    :: columnCount

  character(len=*), intent(in), optional :: label ! label (if present)
                                                  ! otherwise, current
                                                  ! line

  integer, intent(out), optional        :: rc     ! Error code
```

DESCRIPTION:

Returns the number of lines in the table in `lineCount` and the maximum number of words in a table line in `columnCount`.
The arguments are:

config Already created `ESMF_Config` object.

lineCount Returned number of lines in the table.

columnCount Returned maximum number of words in a table line.

[label] Identifying label.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.15 ESMF_ConfigGetLen - Get the length of the line in words

INTERFACE:

```
integer function ESMF_ConfigGetLen( config, label, rc )
```

ARGUMENTS:

```
  type(ESMF_Config), intent(inout)       :: config
  character(len=*), intent(in), optional :: label
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Gets the length of the line in words by counting words disregarding types. Returns the word count as an integer.
The arguments are:

config Already created `ESMF_Config` object.

[label] Identifying label. If not specified, use the current line.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.16 ESMF_ConfigLoadFile - Load resource file into memory

INTERFACE:

```
subroutine ESMF_ConfigLoadFile( config, filename, delayout, unique, rc )
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout) :: config
character(len=*), intent(in)     :: filename
type(ESMF_DELayout), intent(in), optional  :: delayout
logical, intent(in), optional    :: unique
integer, intent(out), optional :: rc
```

DESCRIPTION:

Resource file with `filename` is loaded into memory.
The arguments are:

config Already created `ESMF_Config` object.

filename Configuration file name.

[delayout] `ESMF_DELayout` associated with this `config` object.

[unique] If specified as true, uniqueness of labels are checked and error code set if duplicates found.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

38.3.17 ESMF_ConfigNextLine - Find next line

INTERFACE:

```
subroutine ESMF_ConfigNextLine( config, tableEnd, rc)
```

ARGUMENTS:

```
type(ESMF_Config), intent(inout) :: config
logical, intent(out), optional :: tableEnd
integer, intent(out), optional:: rc
```

DESCRIPTION:

Selects the next line (for tables).
The arguments are:

config Already created ESMF_Config object.

[tableEnd] If specifed as TRUE, end of table mark (::) is checked.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

39 LogErr Class

39.1 Description

The Log class consists of a variety of methods for writing error, warning, and informational messages to files. A default Log is created at ESMF initialization. Other Logs can be created later in the code by the user. A set of standard return codes and associated messages are provided for error handling.
Log provides capabilities to write to a file immediately or store entries in a buffer for writing either when the buffer is full, or when the user calls an ESMF_LogFlush() command. Currently the Log flushes after every write command. Many options are planned for Log, including the ability to write to a single file or multiple files, to write from a root PE or from all active PEs, to halt at errors, at warnings, or never, to adjust the verbosity of ouput, and to optionally write to stdout instead of file(s).

39.2 LogErr Options

39.2.1 ESMF_MsgType

DESCRIPTION:
Specifies what sort of message - e.g., warning, info, error - will be written to an ESMF_Log file.
Valid values are:

ESMF_LOG_INFO Message is informational.

ESMF_LOG_WARNING Message is a warning.

ESMF_LOG_ERROR Message indicates an error.

39.2.2 ESMF_LogType

DESCRIPTION:
Specifies single or multi log.
Valid values are:

ESMF_LOG_SINGLE Log is single log.

ESMF_LOG_MULTI Log is multi log.

39.3 Use and Examples

A default Log is created at ESMF_Initialize(). ESMF handles the initialization and finalization of the default Log so the user can immediately start using it. A single default Log is opened in the directory that initializes the default Log. If a Log is not present, a new one is created. If multiple Logs are desired, they must be explicitly created or opened using ESMF_LogOpen().
If a user wants to use a new or different Log, the user must call ESMF_LogOpen() and pass in a Log object and filename to open a Log file.

By default, the Log file is not truncated at the start of a new run; it just gets appended to each time. Future functionality could include an option to either truncate or append to the Log file.
In all cases where a Log is opened, a unit number is assigned to a specific Log. A Log is assigned the lowest available unit number starting with 11. If a unit number is occupied, the next higher unit number is checked using the "inquire" method. The process repeats until a free unit number is found or when the unit number reaches `ESMF_LOGUPPER` in which case an error is returned. As a result, the user should always check for free numbers using "inquire" to prevent potential unit number conflicts. In the near future we anticipate supporting an option in which a desired unit number can be passed in.
The user can then set or get options on how the Log should be used with the `ESMF_LogSet()` and `ESMF_LogGet()` methods. These are not fully implemented at this time.
Depending on how the options are set, `ESMF_LogWrite()` either writes user messages directly to a Log file or writes to a buffer that can be flushed when full or by using the `ESMF_LogFlush()` method. In the current implementation the Log flushes after every write.
For every `ESMF_LogWrite()`, a time and date stamp is prepended to the Log entry. The time is given in microsecond precision.
When calling `ESMF_LogWrite()`, the user can supply an optional line, file and method. These arguments can be passed in explicitly or with the help of cpp macros. In the latter case, a define for an `ESMF_FILE` must be placed at the beginning of the code and a define for `ESMF_METHOD` must be placed at the beginning of each method. The user can then use the `ESMF_CONTEXT` cpp macro in place of line, file and method to insert the parameters into the method. The user does not have to specify line number as it is a value supplied by cpp.
When done writing messages, the Log is closed by calling `ESMF_LogClose()` which will release the assigned unit number.

```
! !PROGRAM: ESMF_LogErrEx - Log Error examples
!
! !DESCRIPTION:
!
! This program shows examples of Log Error writing
!-----------------------------------------------------------------------------

! Macros for cpp usage
#include "ESMF_LogMacros.inc"
! ESMF Framework module
    use ESMF_Mod
    implicit none

    ! return variables
    integer :: rc1,rc2
    ! function return variables
    logical :: ret
    ! a log object that is not the default log
    type(ESMF_LOG):: alog
```

39.3.1 Default Log

This example shows how to use the default log. This example does not use cpp macros.

```
    ! Initialize ESMF to initialize the default log
    call ESMF_Initialize(rc=rc1)


    ! LogWrite
    ret= ESMF_LogWrite("Log Write 2",ESMF_LOG_INFO)


    ! LogMsgFoundError
```

```
    ret = ESMF_LogMsgFoundError(ESMF_FAILURE,"hello",rcToReturn=rc2)


    ! LogMsgFoundAllocError
    ret = ESMF_LogFoundAllocError(ESMF_FAILURE,rcToReturn=rc2)
```

39.3.2 User Created Log

This example shows how to use a user created log. This example uses cpp macros.

```
! File define
! Method define
#define ESMF_FILE "ESMF_LogErrEx File"
! Method define
#define ESMF_METHOD "ESMF_LogErrEx Method"
    ! Open a log named "Testlog.txt" associated with alog.
    call ESMF_LogOpen(alog, "TestLog.txt",rc=rc1)


    ! LogWrite
    ret= ESMF_LogWrite("Log Write 2", ESMF_LOG_INFO, ESMF_CONTEXT,log=alog)


    ! LogMsgFoundError
    ret = ESMF_LogMsgFoundError(ESMF_FAILURE,"hello", ESMF_CONTEXT, &
          rcToReturn=rc2,log=alog)


    ! LogMsgFoundAllocError
    ret = ESMF_LogFoundAllocError(ESMF_FAILURE, ESMF_CONTEXT, rc2,alog)


    ! Close the log.
    call ESMF_LogClose(alog,rc2)



    ! Finalize ESMF to close the default log
    call ESMF_Finalize(rc=rc1)



end program
```

39.4 Restrictions and Future Work

1. **Line and file only available when using the C preprocessor** Message writing methods are expanded using a macro that adds the predefined symbolic constants __LINE__ and __FILE__ to the argument list. Using these constants, we can associate a file name and line number with the message. If the CPP preprocessor is not used, this expansion will not be done and the file and line number will not appear in the Log text.
2. **Get and set methods not implemented.** Currently, the `ESMF_LogGet()` and `ESMF_LogSet()` methods are not implemented.
3. **Flush after write.** Flushing occurs after every write instead of allowing the user to specify the buffer size.

4. **Log only appends entries.** Currently, all writing to the log is appended rather deleting the old contents and writing to a clean log. Future enhancements include the option to either append to an existing log or write to a clean log.

5. **Avoiding conflicts with the default Log.** The private methods `ESMF_LogInitialize()` and `ESMF_LogFinalize()` are called during `ESMF_Initialize()` and `ESMF_Finalize()` so they do not need to be called if the default Log is used. If a new Log is required, `ESMF_LogOpen()` is used with a new Log object passed in so that there are no conflicts with the default Log.

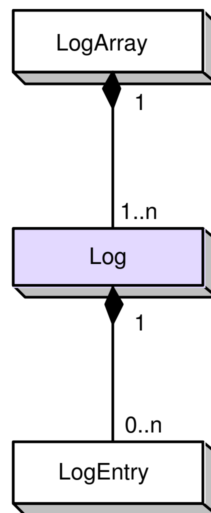
39.5 Design and Implementation Notes

1. The Log class was implemented in C/C++, but uses the Fortran I/O libraries when the class methods are called from Fortran. We forced the C/C++ methods to use the Fortran I/O library by creating utility functions that are written in Fortran, but callable from Log's C++ methods. These utility functions call the standard Fortran write, open and close functions. If you call the Log methods from C/C++ code, you bypass the utility functions and all I/O is done with the C I/O libraries. At initialization a LogArray is created. This array is a container that holds multiple Logs. Within the LogArray, one or more Logs may be created. Each Log stores information for a specific Log file. When working with more than one Log file, multiple Logs are required (one Log for each Log file). For each Log, a handle is returned through the `ESMF_LogOpen()` method so that the user can manage the various Logs. The properties of a specific Log are set with the `ESMF_LogSet()` method. Additionally, buffering occurs within a Log. Every time the `ESMF_LogWrite()` method is called, LogEntry element is populated with the `ESMF_LogWrite()` information. The user can set the number of LogEntry elements in the Log buffer. When the buffer is full (i.e., when all the LogEntry elements are populated), the buffer will be flushed and all the contents will be written to file. If buffering is not needed (`autoflush=false`), the `ESMF_LogWrite()` method will immediately write to the Log files.

   Buffering allows ESMF to manage output data streams in a desired way. For instance, a solution to clean streaming PE_ordered output would be to gather the buffers on root PE when flushing and order and stream them as desired by the user. Buffering the output provides a layer where the output can be managed. Writing to the buffer is transparent to the user because all the Log entries are handled automatically by the `ESMF_LogWrite()` method.

39.6 Object Model

The following is a simplified UML diagram showing the structure of the Log class. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



39.7 Class API

39.7.1 ESMF_LogClose - Close Log file(s)

INTERFACE:

```
subroutine ESMF_LogClose(log, rc)
```

ARGUMENTS:

```
type(ESMF_Log)              :: log
integer, intent(out),optional   :: rc
```

DESCRIPTION:

This routine closes the file(s) associated with the `log`.
The arguments are:

log An `ESMF_Log` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

39.7.2 ESMF_LogFoundAllocError - Check Fortran status for allocation error

INTERFACE:

```
function ESMF_LogFoundAllocError(statusToCheck, line, file, &
                                        method, rcToReturn,log)
```

RETURN VALUE:

```
logical                                   ::ESMF_LogFoundAllocError
```

ARGUMENTS:

```
integer, intent(in)                       :: statusToCheck
integer, intent(in), optional             :: line
character(len=*), intent(in), optional    :: file
character(len=*), intent(in), optional    :: method
integer, intent(out),optional             :: rcToReturn
type(ESMF_LOG),intent(in),optional :: log
```

DESCRIPTION:

This function returns a logical true when a Fortran status code returned from a memory allocation indicates an allocation error. An ESMF predefined memory allocation error message will be added to the `ESMF_Log` along with `line`, `file` and `method`. Additionally, the `statusToCheck` will be converted to a `rcToReturn`.
The arguments are:

statusToCheck Fortran allocation status to check.

[line] Integer source line number. Expected to be set by using the preprocessor macro `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, set the rcToReturn value to ESMF_RC_MEM which is the error code for a memory allocation eror.

[log] An optional ESMF_Log object that can be used instead of the default log.

39.7.3 ESMF_LogFoundError - Check ESMF return code for error

INTERFACE:

```
function ESMF_LogFoundError(rcToCheck, line, file, method,&
         rcToReturn, log)
```

RETURN VALUE:

```
logical                                          ::ESMF_LogFoundError
```

ARGUMENTS:

```
integer, intent(in)                              :: rcToCheck
integer, intent(in), optional                    :: line
character(len=*), intent(in), optional           :: file
character(len=*), intent(in), optional          :: method
integer, intent(out), optional                   :: rcToReturn
type(ESMF_LOG),intent(in),optional :: log
```

DESCRIPTION:

This function returns a logical true for ESMF return codes that indicate an error. A predefined error message will added to the ESMF_Log along with line, file and method. Additionally, rcToReturn will be set to rcToCheck. The arguments are:

rcToCheck Return code to check.

[line] Integer source line number. Expected to be set by using the preprocessor macro __LINE__ macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, copy the rcToCheck value to rc. This is not the return code for this function; it allows the calling code to do an assignment of the error code at the same time it is testing the value.

[log] An optional ESMF_Log object that can be used instead of the default log.

39.7.4 ESMF_LogMsgFoundAllocError - Check Fortran status for allocation error and write message

INTERFACE:

```
function ESMF_LogMsgFoundAllocError(statusToCheck,msg,line,file, &
                                    method,rcToReturn,log)
```

RETURN VALUE:

```
logical                              ::ESMF_LogMsgFoundAllocError
```

ARGUMENTS:

```
integer, intent(in)                          :: statusToCheck
character(len=*), intent(in)                 :: msg
integer, intent(in), optional                :: line
character(len=*), intent(in), optional       :: file
character(len=*), intent(in), optional :: method
       integer, intent(out),optional           :: rcToReturn
       type(ESMF_LOG), intent(in), optional :: log
```

DESCRIPTION:

This function returns a logical true when a Fortran status code returned from a memory allocation indicates an allocation error. An ESMF predefined memory allocation error message will be added to the `ESMF_Log` along with a user added `msg`, `line`, `file` and `method`. Additionally, statusToCheck will be converted to `rcToReturn`.
The arguments are:

[statusToCheck] Fortran allocation status to check.

[msg] User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor macro __`LINE`__ macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, set the `rcToReturn` value to `ESMF_RC_MEM` which is the error code for a memory allocation eror.

[log] An optional `ESMF_Log` object that can be used instead of the default log.

39.7.5 ESMF_LogMsgFoundError - Check ESMF return code for error and write message

INTERFACE:

```
function ESMF_LogMsgFoundError(rcToCheck, msg, line, file, method, &
                                        rcToReturn, log)
```

RETURN VALUE:

```
logical                                            ::ESMF_LogMsgFoundError
```

ARGUMENTS:

```
integer, intent(in)                                :: rcToCheck
character(len=*), intent(in)                       :: msg
integer, intent(in), optional                      :: line
character(len=*), intent(in), optional             :: file
character(len=*), intent(in), optional            :: method
integer, intent(out),optional                      :: rcToReturn
type(ESMF_LOG), intent(in), optional :: log
```

DESCRIPTION:

This function returns a logical true for ESMF return codes that indicate an error. A predefined error message will added to the ESMF_Log along with a user added msg, line, file and method. Additionally, rcToReturn will be set to rcToCheck.
The arguments are:

rcToCheck Return code to check.

msg User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor macro __LINE__ macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, copy the rcToCheck value to rc. This is not the return code for this function; it allows the calling code to do an assignment of the error code at the same time it is testing the value.

[log] An optional ESMF_Log object that can be used instead of the default log.

39.7.6 ESMF_LogOpen - Open Log file(s)

INTERFACE:

```
subroutine ESMF_LogOpen(log, filename, rc)
```

ARGUMENTS:

```
type(ESMF_Log)      :: log
character(len=*)      :: filename
integer, intent(out),optional      :: rc
```

DESCRIPTION:

This routine opens a file with filename and associates it with the log. This is only used when the user does not want to use the default ESMF_Log.
The arguments are:

[log] An ESMF_Log object.

[filename] Name of file.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

39.7.7 ESMF_LogWrite - Write to Log file(s)

INTERFACE:

```
function ESMF_LogWrite(msg,MsgType,line,file,method,log)
```

RETURN VALUE:

```
  logical                                    ::ESMF_LogWrite
```

ARGUMENTS:

```
  character(len=*), intent(in)               :: msg
  type(ESMF_MsgType), intent(in)             :: msgtype
  integer, intent(in), optional              :: line
  character(len=*), intent(in), optional     :: file
  character(len=*), intent(in), optional :: method
  type(ESMF_LOG), intent(in), optional :: log
```

DESCRIPTION:

This function writes to the file associated with an `ESMF_Log`. A message is passed in along with the `msgtype`, `line`, `file` and `method`. If the write to the `ESMF_Log` is successful, the function will return a logical `true`. This function is the base function used by all the other `ESMF_Log` writing methods.
The arguments are:

msg User-provided message string.

msgtype The type of message. See Section 39.2.1 for possible values.

[line] Integer source line number. Expected to be set by using the preprocessor macro `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[log] An optional `ESMF_Log` object that can be used instead of the default log.

40 DELayout Class

40.1 Description

The DELayout class provides an additional layer of abstraction on top of the Virtual Machine (VM) layer. There are three key aspects the DELayout class deals with.

1. Problem decomposition via logical Decomposition Elements (DEs).
2. Support of load balancing in terms of computational and connection weights on and between the DEs.
3. Mapping of the logical problem decomposition onto an ESMF Virtual Machine.

It is critical to understand that *no* user data is associated with DELayout objects! DELayouts are *control* objects which store important decomposition information and provide crucial functionality to ESMF applications with respect to various aspects of logical problem decomposition. Data objects, such as Arrays and Fields, which hold user data, rely in the implementation of their methods on the DELayout class to provide this decomposition functionality.
The application writer uses the DELayout to specify the decomposition of the computational problem in terms of logical Decomposition Elements (DEs). From an ESMF perspective the DEs are the smallest units of decomposition. DEs are logical units, not necessarily having a 1-to-1 correspondence to the Persistent Execution threads (PETs) of a VM or their physical Processing Elements (PEs) in the underlying physical machine. Consequently there are no restrictions on the number of DEs possed by the VM or the available physical machine resources. Hence, the application writer may chose the number of DEs to best match the computational problem and the employed algorithm.
A DELayout object not only keeps track of the number of DEs into which a problem is decomposed, but furthermore allows the user to specify a problem topology by means of computational weights on each DE and connection weights between DEs. Both types of weights are relative measures, meaningful only for comparison *within* the same DELayout object. The purpose of these weights is to provide load balancing information and to allow for a best possible DE-to-PET mapping of a DELayout onto the component's VM.

It is possible for the application writer to overwrite the framework's DE-to-PET mapping. This allows for user level load balancing schemes and offers an entry point for user codes that already deal with the issue of mapping the *computational problem topology* onto the *resource topology*.
In a typical ESMF application direct interaction with the DELayout class is minimal. This is reflected by the small number of methods in the DELayout API. Besides the `ESMF_DELayoutCreate()` and `ESMF_DELayoutDestroy()` methods there are three types of `ESMF_DELayoutGet` methods. After creating a DELayout the application writer will use the get methods to find out general information, such as number of DEs and dimensionality of the decomposition. More importantly though, because it is unknown until the DELayout has been created, is the *PET-local* information that is provided by the DELayout Get methods. Having obtained PET specific information about the decomposition each running user thread can take appropriate steps in facilitating the decomposition within its local PET.
Notice that a single ESMF component may contain multiple DELayouts, all of which may describe the decomposition of different computational problems or different compositions of the same computational problem. However, all DELayouts within a component map onto the same VM instance of the component.

40.2 Use and Examples

The following examples demonstrate how to create, use and destroy logically rectangular DELayouts.

40.2.1 Default 1-D DELayout

Without additional parameters the created `ESMF_DELayout` will default into a 1-dimensional DELayout with as many DEs as there are PETs in the associated VM object. Consequently the resulting DELayout will always be 1-to-1, i.e. each DE maps onto exactly one PET of the VM.

```
delayout = ESMF_DELayoutCreate(vm, rc=rc)


call ESMF_DELayoutPrint(delayout, rc=rc)


call ESMF_DELayoutDestroy(delayout, rc=rc)
```

40.2.2 1-D DELayout with Fixed Number of DEs

The `deCountList` argument has two functions when present. First it specifies the total number of DEs and second it specifies the dimensionality of the DELayout. Here a 1-dimensional DELayout will be created with 4 DEs. Note that it depends on the VM whether this will be a 1-to-1 DELayout or not. If the VM contains 4 PETs or more the DELayout will be 1-to-1, otherwise there will be virtual DEs present.

```
delayout = ESMF_DELayoutCreate(vm, deCountList=(/4/), rc=rc)


call ESMF_DELayoutPrint(delayout, rc=rc)


call ESMF_DELayoutGet(delayout, oneToOneFlag=otoflag, rc=rc)


if (otoflag==ESMF_TRUE) then
  print *, 'This is a 1-to-1 DELayout'
else
  print *, 'This is a DELayout with virutal DEs'
endif

call ESMF_DELayoutDestroy(delayout, rc=rc)
```

40.2.3 2-D DELayout with Fixed Number of DEs

Here a 2-dimensional DELayout will be created with 6 DEs, layed out as 2x3. As in the previous example it depends on the VM whether this will be a 1-to-1 DELayout or not. If the VM contains 6 PETs or more the DELayout will be 1-to-1, otherwise there will be virtual DEs present.

```
delayout = ESMF_DELayoutCreate(vm, deCountList=(/2, 3/), rc=rc)
```

```
call ESMF_DELayoutPrint(delayout, rc=rc)
```

```
call ESMF_DELayoutDestroy(delayout, rc=rc)
```

40.3 Restrictions and Future Work

1. **Virtual DE capabilities not to be used.** Only 1-to-1 DELayouts are supported by other parts of the ESMF library.
2. **Logically rectangular DELayouts only.** Only the logical rectangular DELayout creation method is currently implemented.
3. **Computational weights are not yet implemented.** This is anticipated in the near term.
4. **Load balancing functionality is not yet implemented.** This capability will be implemented along with compuatational weights per DE.

40.4 Design and Implementation Notes

The DELayout class stores information about the DEs and their connectivity, thus holding an abstraction of the computational problem. Furthermore a DELayout object holds DE-to-PET mapping info, which maps *every* DE to *exactly one* PET of the underlying VM. The DELayout is associated with the VM of the context in which the DELayout was created.

40.5 Class API

40.5.1 ESMF_DELayoutCreate - Create N-dimensional logically rectangular DELayout

INTERFACE:

```
! Private name; call using ESMF_DELayoutCreate()
function ESMF_DELayoutCreateND(vm, deCountList, dePetList, &
  connectionWeightDimList, cyclicFlagDimList, rc)
```

ARGUMENTS:

```
type(ESMF_VM),       intent(in)                :: vm
integer, target,     intent(in),   optional  :: deCountList(:)
integer, target,     intent(in),   optional  :: dePetList(:)
integer,             intent(in),   optional  :: connectionWeightDimList(:)
type(ESMF_Logical), intent(in),   optional  :: cyclicFlagDimList(:)
integer,             intent(out),  optional  :: rc
```

RETURN VALUE:

```
type(ESMF_DELayout) :: ESMF_DELayoutCreateND
```

DESCRIPTION:

Create an N-dimensional, logically rectangular ESMF_DELayout. Depending on the optional argument deCountList there are two cases that can be distinguished:

- If deCountList is missing the method will create a 1-dimensional 1:1 DE-to-PET layout with as many DEs as there are PETs in the VM.
- If deCountList is present the method will create an N-dimensional layout, where N is equal to the the size of deCountList. The number of DEs will be deCountList(1) × deCountList(2) ×...× deCountList(N).

In either case, if dePetList is given and its size is equal to the number of DEs in the created ESMF_DELayout, it will be used to determine the DE-to-PET mapping. Otherwise a DE-to-PET mapping will be determined automatically. The connectionWeightDimList argument, if present, must have N entries which will be used to ascribe connection weights along each dimension within the ESMF_DELayout. These weights have values from 0 to 100 and will be used to find the best match between an ESMF_DELayout and the ESMF_VM.
The cyclicFlagDimList argument allows to enforce cyclic boundaries in each of the dimensions of ESMF_DELayout. If present its size must be equal to the number of DEs in the ESMF_DELayout. (*Not yet implemented feature!*)

The arguments are:

vm ESMF_VM object of the current component in which the ESMF_DELayout object shall operate.

[deCountList] List DE count in each dimension.

[dePetList] List specifying DE-to-PET mapping.

[connectionWeightDimList] List of connection weights along each dimension.

[cyclicFlagDimList] List of flags indicating cyclic boundaries in each dimension. (*Not yet implemented feature!*)

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

40.5.2 ESMF_DELayoutDestroy - Destroy DELayout object

INTERFACE:

```
subroutine ESMF_DELayoutDestroy(delayout, rc)
```

ARGUMENTS:

```
  type(ESMF_DELayout),  intent(in)              :: delayout
  integer,              intent(out),  optional  :: rc
```

DESCRIPTION:

Destroy an ESMF_DELayout object.
The arguments are:

delayout ESMF_DELayout object to be destroyed.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

40.5.3 ESMF_DELayoutGet - Get DELayout internals

INTERFACE:

```
subroutine ESMF_DELayoutGet(delayout, deCount, dimCount, localDeCount, &
  localDeList, localDe, oneToOneFlag, logRectFlag, deCountPerDim, rc)
```

ARGUMENTS:

```
type(ESMF_DELayout),  intent(in)             :: delayout
integer,              intent(out),  optional  :: deCount
integer,              intent(out),  optional  :: dimCount
integer,              intent(out),  optional  :: localDeCount
integer, target,      intent(out),  optional  :: localDeList(:)
integer,              intent(out),  optional  :: localDe
type(ESMF_Logical),   intent(out),  optional  :: oneToOneFlag
type(ESMF_Logical),   intent(out),  optional  :: logRectFlag
integer, target,      intent(out),  optional  :: deCountPerDim(:)
integer,              intent(out),  optional  :: rc
```

DESCRIPTION:

Get internal decomposion information.
The arguments are:

delayout Queried `ESMF_DELayout` object.

[deCount] Upon return this holds the total number of DEs.

[dimCount] Upon return this holds the number of dimensions in the specified `ESMF_DELayout` object's coordinate tuples.

[localDeCount] Upon return this holds the number of DEs associated with the local PET.

[localDeList] Upon return this holds the list of DEs associated with the local PET.

[localDe] If the specified `ESMF_DELayout` object is 1-to-1 then upon return this holds the DE associated with the local PET.

[oneToOneFlag] Upon return this holds `ESMF_TRUE` if the specified `ESMF_DELayout` object is 1-to-1, `ESMF_FALSE` otherwise.

[logRectFlag] Upon return this holds `ESMF_TRUE` if the specified `ESMF_DELayout` object is logically rectangular, `ESMF_FALSE` otherwise.

[deCountPerDim] If the specified `ESMF_DELayout` object is logically rectangular then upon return this holds the number of DEs along each dimension.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

40.5.4 ESMF_DELayoutGetDELocalInfo - Get DE specific DELayout internals

INTERFACE:

```
subroutine ESMF_DELayoutGetDELocalInfo(delayout, de, coord, connectionCount, &
  connectionList, connectionWeightList, rc)
```

ARGUMENTS:

```
type(ESMF_DELayout),  intent(in)              :: delayout
integer,              intent(in)              :: de
integer, target,      intent(out),  optional  :: coord(:)
integer,              intent(out),  optional  :: connectionCount
integer, target,      intent(out),  optional  :: connectionList(:)
integer, target,      intent(out),  optional  :: connectionWeightList(:)
integer,              intent(out),  optional  :: rc
```

DESCRIPTION:

Get DE specific internal information about the decomposition.
The arguments are:

delayout Queried `ESMF_DELayout` object.

de Queried DE id within the specified `ESMF_DELayout` object.

[coord] Upon return this holds the coordinate tuple of the specified DE.

[connectionCount] Upon return this holds the number of connections associated with the specified DE.

[connectionList] Upon return this holds the list of DEs the specified DE is connected to.

[connectionWeightList] Upon return this holds the list of connection weights of all the connections with the specified DE.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

40.5.5 ESMF_DELayoutGetDEMatch - Match virtual memory spaces between DELayouts

INTERFACE:

```
subroutine ESMF_DELayoutGetDEMatch(delayout, de, delayoutMatch, &
  deMatchCount, deMatchList, rc)
```

ARGUMENTS:

```
type(ESMF_DELayout),  intent(in)              :: delayout
integer,              intent(in)              :: de
type(ESMF_DELayout),  intent(in)              :: delayoutMatch
integer,              intent(out),  optional  :: deMatchCount
integer, target,      intent(out),  optional  :: deMatchList(:)
integer,              intent(out),  optional  :: rc
```

DESCRIPTION:

Match the virtual memory space of the specified DE in a DELayout with that of the DEs of a second DELayout. The use of this method is crutial when dealing with decomposed data structures that were not defined in the current VM context, i.e. defined in another component.
The arguments are:

delayout `ESMF_DELayout` object in which the specified DE is defined.

de Specified DE within delayout, for which to find matching DEs in delayoutMatch,

delayoutMatch DELayout object in which to find DEs that match the virtual memory space of the specified DE.

[deMatchCount] Upon return this holds the number of DEs in delayoutMatch that share virtual memory space with the specified DE.

[deMatchList] Upon return this holds the list of DEs in delayoutMatch that share virtual memory space with the specified DE.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

40.5.6 ESMF_DELayoutPrint - Print DELayout internals

INTERFACE:

```
   subroutine ESMF_DELayoutPrint(delayout, options, rc)
```

ARGUMENTS:

```
     type(ESMF_DELayout),  intent(in)              :: delayout
     character(len=*),     intent(in),   optional  :: options
     integer,              intent(out),  optional  :: rc
```

DESCRIPTION:

Prints internal information about the specified `ESMF_DELayout` object to `stdout`.
The arguments are:

delayout Specified `ESMF_DELayout` object.

[options] Print options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41 VM Class

41.1 Description

The `ESMF_VM` (Virtual Machine) class can be viewed as a generic representation of hardware and system software resources. There is exactly one VM object associated with each component in an ESMF application. The VM object handles all resource management tasks of a component and provides a topological description of the underlying configuration of the compute resources used by the component. The basic elements of a VM are called PETs, which stands for Persistent Execution Threads. These are equivalent to OS threads with a lifetime of at least that of the associated component. All VM functionality is expressed in terms of PETs. In the single-threaded case - the only one currently accessible in ESMF - a PET is equivalent to a MPI process.
The resource management functions of the VM class come into play when a component creates sub-components. There are two parts to resource management, the parent and the child. When the parent component creates a child component it provides its own VM object to the `ESMF_GridCompCreate()` or `ESMF_CplCompCreate()` method and optionally specifies a `petList` to limit the resources it wants to give to the child. The child on the other hand may specify - during its SetServices method - how it wants the inherited resources to be arranged in its own VM. Besides the SetServices method all other registered component methods will henceforth be instantiated using the thus defined child VM.
In addition to resource management and topological description the VM class offers the lowest level of ESMF communication methods. Data references in VM communication calls must be provided as raw, language specific, one-dimensional, contiguous data arrays, much like in MPI. In fact, the similarity between VM and MPI communication

calls is striking and there are many equivalent point-to-point and collective communication calls. However, two major differences set VM communication apart from its MPI counterpart. First, the transparent VM communication calls hide an array of very specific implementations, ranging from intra-process communication within multi-threaded processes, shared memory communication between processes within the same single system image, to the use of specific hardware libraries associated with the interconnection fabric in distributed systems. Second, unlike MPI, the VM class provides non-blocking collective communication calls in addition to the non-blocking point-to-point primitives.

41.2 Use and Examples

The concept of the ESMF Virtual Machine (VM) is so fundamental to the framework that every ESMF application uses it. Even in the simplest case, that of an ESMF main program without any components, a global default VM is being created during the `ESMF_Initialize()` call and removed during `ESMF_Finalize()`.
By its very nature the VM class is quite different from other ESMF classes. One reflection of this fact is that VM objects appear in the API of infrastructure *and* superstructure ESMF classes. The first place to encounter a VM object is at the `ESMF_Initialize()` call. If the optional `vm=` argument is specified the global default VM will be returned to the user code. The default VM can also be obtained throughout the application by calling `ESMF_VMGetGlobal()`. The default VM is an MPI-only VM that spans all processes in `MPI_COMM_WORLD` and it is the context in which the main program is executing. After the initialization the default VM may be used within the main program in query or communication calls.
One of the main tasks of the VM class is resource management. Thus the VM plays a major part when a new ESMF component is created. On the parent side of this process the parent VM serves as a contributor of resources. When the parent component creates a child component it provides its own VM object and further may specify a list of resources (in terms of PETs) that it wants to give to the child component. This allows a parent to divide its resources among several children without oversubscribing the computational resources it holds.
On the child side of the creation process each child may set key properties of its VM, i.e. it is up to the child component to decide on how to use the resources it receives from the parent component. In the current version of ESMF multi-threading has been deactivated and only the default MPI-only VM option is supported.
After a child component is created by a parent it may be entered via one of the registered initialize / run / finalize entry points. Each time a component is entered through these methods the associated component routine will start running in the context of its own VM. The user code may gain access to the VM of its context by querying the active component object. Once obtained the VM may be used in query and communication calls, and - creating a hierarchy of components - to create child components.

41.2.1 VM Default Basics Example

This complete example program demonstrates the simplest ESMF application, consisting of only a main program without any components. The global default VM, which is automatically created during the `ESMF_Initialize()` call, is obtained and then used in its print method and several VM query calls.

```
program ESMF_VMDefaultBasicsEx

  use ESMF_Mod

  implicit none

  ! local variables
  integer:: rc
  type(ESMF_VM):: vm
  integer:: localPet, petCount, peCount, ssiId


  call ESMF_Initialize(vm=vm, rc=rc)


  call ESMF_VMPrint(vm, rc=rc)
```

```
  call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, peCount=peCount, &
    rc=rc)


  print *, 'localPet is: ', localPet,' out of a total of ',petCount,' PETs.'
  print *, 'there are ', peCount,' PEs referenced by this VM'

  call ESMF_VMGetPETLocalInfo(vm, localPet, peCount=peCount, ssiId=ssiId, rc=rc)


  print *, 'localPet is: ', localPet,' and it is claiming ', peCount, &
    ' PEs on SSI ', ssiId

  call ESMF_Finalize(rc=rc)


end program
```

41.2.2 VMGet MPI Communicator Example

The following example code shows how to obtain the MPI intra-communicator out of a VM object. In order not to interfere with ESMF communications it is advisable to duplicate the communicator before using it in user-level MPI calls. In this example the duplicated communicator is used for a user controlled barrier accross the context.

```
  integer:: mpic, mpic2


  call ESMF_VMGet(vm, mpiCommunicator=mpic, rc=rc)


  call MPI_Comm_Dup(mpic, mpic2, ierr)
  call MPI_Barrier(mpic2, ierr)
```

41.2.3 VMSend/VMRecv Example

The VM layer provides MPI-like point-to-point communication. Use VMSend and VMRecv to communicate between two PETs. The following SPMD code sends data from PET 'src' and receives it on PET 'dst' of the VM. The sendData and recvData arguments must be 1-dimensional arrays.

```
  if (localPet==src) &
    call ESMF_VMSend(vm, sendData=localData, count=count, dst=dst, rc=rc)


  if (localPet==dst) &
    call ESMF_VMRecv(vm, recvData=localData, count=count, src=src, rc=rc)
```

41.2.4 VMScatter/VMGather Example

The VM layer provides MPI-like collective communication. This example demonstrates the use of VM-wide VM-Scatter and VMGather.

```
  call ESMF_VMScatter(vm, sendData=array1, recvData=array2, count=nsize, &
    root=scatterRoot, rc=rc)


  call ESMF_VMGather(vm, sendData=array2, recvData=array1, count=nsize, &
    root=gatherRoot, rc=rc)
```

41.2.5 VMAllFullReduce Example

The VMAllFullReduce method can be used to find the VM-wide global sum of a data set.

```
  call ESMF_VMAllFullReduce(vm, sendData=array1, recvData=result, count=nsize, &
    reduceflag=ESMF_SUM, rc=rc)
```

41.2.6 VM Component Example

The following example shows the role that VMs play in connetion with ESMF components. Here a single component is created in the main program and the default VM gives all its resources to the child component. When the child component code is entered through the registered methods (Initialize, Run or Finalize) the user code will be executed in the child's VM.

```
module ESMF_VMComponentEx_gcomp_mod


  public mygcomp_register

  contains !--------------------------------------------------------------

  subroutine mygcomp_register(gcomp, rc)


    ! register INIT method
    call ESMF_GridCompSetEntryPoint(gcomp, ESMF_SETINIT, mygcomp_init, &
      ESMF_SINGLEPHASE, rc)
    ! register RUN method
    call ESMF_GridCompSetEntryPoint(gcomp, ESMF_SETRUN, mygcomp_run, &
      ESMF_SINGLEPHASE, rc)
    ! register FINAL method
    call ESMF_GridCompSetEntryPoint(gcomp, ESMF_SETFINAL, mygcomp_final, &
      ESMF_SINGLEPHASE, rc)
  end subroutine !--------------------------------------------------------------

  recursive subroutine mygcomp_init(gcomp, istate, estate, clock, rc)


    ! get this component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    call ESMF_VMPrint(vm, rc)


  end subroutine !--------------------------------------------------------------

  recursive subroutine mygcomp_run(gcomp, istate, estate, clock, rc)
```

```
    ! get this component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    call ESMF_VMPrint(vm, rc)

  end subroutine !--------------------------------------------------------------

  recursive subroutine mygcomp_final(gcomp, istate, estate, clock, rc)

    ! get this component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    call ESMF_VMPrint(vm, rc)

  end subroutine !--------------------------------------------------------------

end module


program ESMF_VMComponentEx


  use ESMF_VMComponentEx_gcomp_mod


  gcomp = ESMF_GridCompCreate(vm, 'My gridded component', rc=rc)


  call ESMF_GridCompSetServices(gcomp, mygcomp_register, rc)


  call ESMF_GridCompInitialize(gcomp, dummystate, dummystate, clock, rc=rc)


  call ESMF_GridCompRun(gcomp, dummystate, dummystate, clock, rc=rc)


  call ESMF_GridCompFinalize(gcomp, dummystate, dummystate, clock, rc=rc)


  call ESMF_GridCompDestroy(gcomp, rc=rc)


  call ESMF_Finalize(rc=rc)


end program
```

41.3 Restrictions and Future Work

1. **Threading turned off.** The multi-threading features of VM are currently disabled in order to prevent conflicts with those parts of ESMF that are not thread-safe.

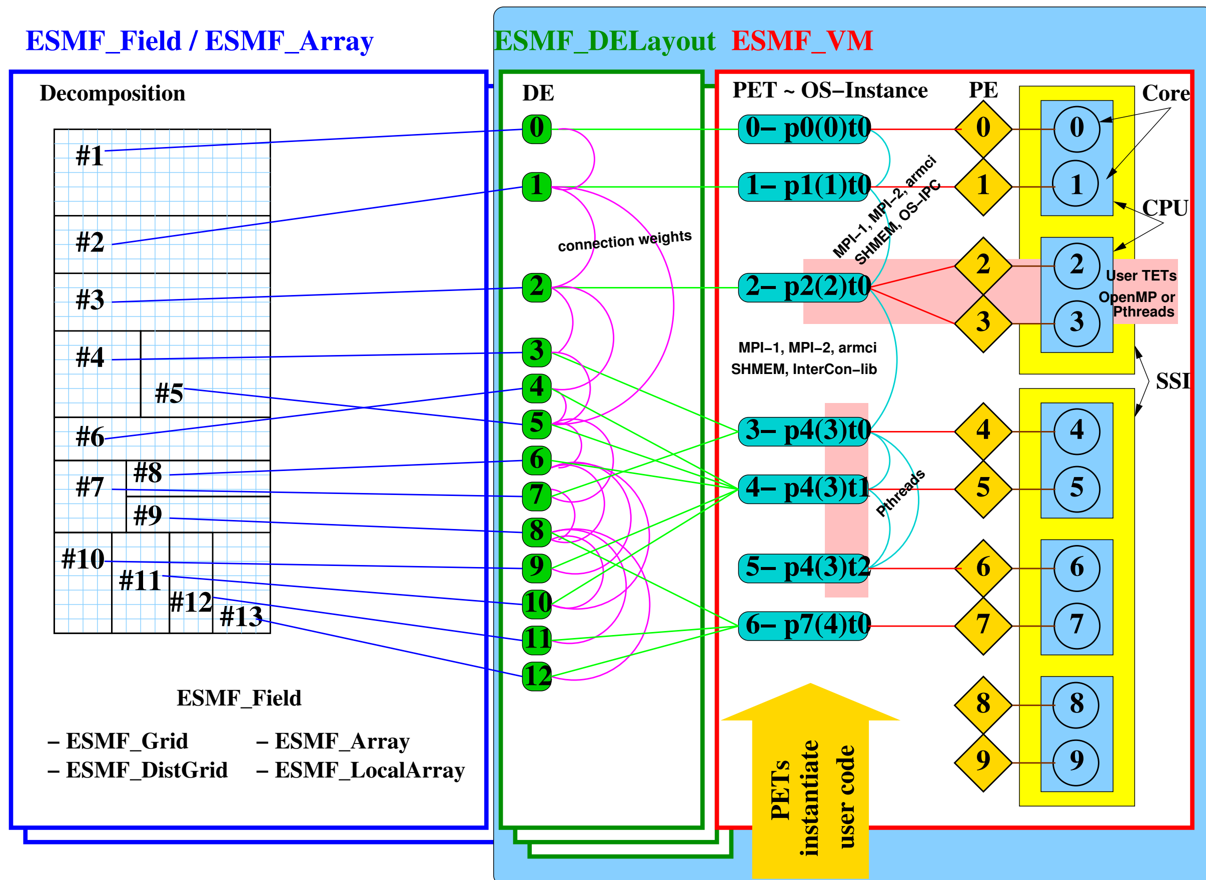
2. **Concurrency support turned off.** The concurrency features of VM are currently disabled.

3. **Nonblocking versions not implemented.** Only the blocking versions of the communication primitives are currently implemented.
4. ESMF_SUM **is the only reduction operation currently implemented.**
5. **None of the topology features have been implemented.**

41.4 Design and Implementation Notes

The VM class provides an additional layer of abstraction on top of the POSIX machine model, making it suitable for HPC applications. There are four key aspects the VM class deals with.

1. Encapsulation of hardware and operating system details within the concept of Persistent Execution Threads (PETs).
2. Resource management in terms of PETs with a guard against over-subscription.
3. Topological description of the underlying configuration of the compute resources in terms of PETs.
4. Transparent communication API for point-to-point and collective PET-based primitives, hiding the many different communication channels and offering best possible performance.



Definition of terms used in the diagram

- PE: A processing element (PE) is an alias for the smallest physical processing unit available on a particular hardware platform. In the language of today's microprocessor architecture technology a PE is identical to a

core, however, if future microprocessor designs change the smallest physical processing unit the mapping of the PE to actual hardware will change accordingly. Thus the PE layer separates the hardware specific part of the VM from the hardware-independent part. Each PE is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.

- Core: A Core is the smallest physical processing unit which typically comprises a register set, an integer arithmetic unit, a floating-point unit and various control units. Traditionally there was one core per CPU, however, today some dual-core CPUs are available and multi-core CPU designs are on most manufacturers' road-maps. Each Core is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- CPU: The central processing unit (CPU) houses single or multiple cores, providing them with the interface to system memory, interconnects and IO. Typically the CPU provides some level of caching for the instruction and data streams in and out of the Cores. Cores in a multi-core CPU typically share some caches. Each CPU is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- SSI: A single system image (SSI) spans all the CPUs controlled by a single running instance of the operating system. SMP and NUMA are typical multi-CPU SSI architectures. Each SSI is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- TOE: A thread of execution (TOE) executes an instruction sequence. TOE's come in two flavors: PET and TET.
- PET: A persistent execution thread (PET) executes an instruction sequence on an associated set of data. The PET has a lifetime at least as long as the associated data set. In ESMF the PET is the central concept of abstraction provided by the VM class. The PETs of an VM object are labeled from 0 to N-1 where N is the total number of PETs in the VM object.
- TET: A transient execution thread (TET) executes an instruction sequence on an associated set of data. A TET's lifetime might be shorter than that of the associated data set.
- OS-Instance: The OS-Instance of a TOE describes how a particular TOE is instantiated on the OS level. Using POSIX terminology a TOE will run as a single thread within a single- or multi-threaded process.
- Pthreads: Communication via the POSIX Thread interface.
- MPI-1, MPI-2: Communication via MPI standards 1 and 2.
- armci: Communication via the aggregate remote memory copy interface.
- SHMEM: Communication via the SHMEM interface.
- OS-IPC: Communication via the operating system's inter process communication interface. Either POSIX IPC or System V IPC.
- InterCon-lib: Communication via the interconnect's library native interface. An example is the Elan library for Quadrics.

The POSIX machine abstraction, while a very powerful concept, needs augmentation when applied to HPC applications. Key elements of the POSIX abstraction are processes, which provide virtually unlimited resources (memory, I/O, sockets, ...) to possibly multiple threads of execution. Similarly POSIX threads create the illusion that there is virtually unlimited processing power available to each POSIX process. While the POSIX abstraction is very suitable for many multi-user/multi-tasking applications that need to share limited physical resources, it does not directly fit the HPC workload where over-subscription of resources is one of the most expensive modes of operation.
ESMF's virtual machine abstraction is based on the POSIX machine model but holds additional information about the available physical processing units in terms of Processing Elements (PEs). A PE is the smallest physical processing unit and encapsulates the hardware details (Cores, CPUs and SSIs).
There is exactly one physical machine layout for each application, and all VM instances have access to this information. The PE is the smallest processing unit which, in today's microprocessor technology, corresponds to a single Core. Cores are arranged in CPUs which in turn are arranged in SSIs. The setup of the physical machine layout is part of the ESMF initialization process.

On top of the PE concept the key abstraction provided by the VM is the PET. All user code is executed by PETs while OS and hardware details are hidden. The VM class contains a number of methods which allow the user to prescribe how the PETs of a desired virtual machine should be instantiated on the OS level and how they should map onto the hardware. This prescription is kept in a private virtual machine plan object which is created at the same time the associated component is being created. Each time component code is entered through one of the component's registered top–level methods (Initialize/Run/Finalize), the virtual machine plan along with a pointer to the respective user function is used to instantiate the user code on the PETs of the associated VM in form of single- or multi-threaded POSIX processes.
The process of starting, entering, exiting and shutting down a VM is very transparent, all spawning and joining of threads is handled by VM methods "behind the scenes". Furthermore, fundamental synchronization and communication primitives are provided on the PET level through a uniform API, hiding details related to the actual instantiation of the participating PETs.
Within a VM object each PE of the physical machine maps to 0 or 1 PETs. Allowing unassigned PEs provides a means to prevent over-subscription between multiple concurrently running virtual machines. Similarly a maximum of one PET per PE prevents over-subscription within a single VM instance. However, over-subscription is possible by subscribing PETs from different virtual machines to the same PE. This type of over-subscription can be desirable for PETs associated with IO work loads expected to be used infrequently and to block often on IO requests.
On the OS level each PET of a VM object is represented by a POSIX thread (Pthread) either belonging to a single– or multi–threaded process and maps to at least 1 PE of the physical machine, ensuring its execution. Mapping a single PET to multiple PEs provides resources for user–level multi–threading, in which case the user code inquires how many PEs are associated with its PET and if there are multiple PEs available the user code can spawn an equal number of threads (e.g. OpenMP) without risking over-subscription. Typically these user spawned threads are short-lived and used for fine-grained parallelization in form of TETs. All PEs mapped against a single PET must be part of a unique SSI in order to allow user–level multi–threading!
In addition to discovering the physical machine the ESMF initialization process sets up the default global virtual machine. This VM object, which is the ultimate parent of all VMs created during the course of execution, contains as many PETs as there are PEs in the physical machine. All of its PETs are instantiated in form of single-threaded MPI processes and a 1:1 mapping of PETs to PEs is used for the default global VM.
The VM design and implementation is based on the POSIX process and thread model as well as the MPI-1.2 standard. As a consequence of the latter standard the number of processes is static during the course of execution and is determined at start-up. The VM implementation further requires that the user starts up the ESMF application with as many MPI processes as there are PEs in the available physical machine using the platform dependent mechanism to ensure proper process placement.
All MPI processes participating in a VM are grouped together by means of an MPI_Group object and their context is defined via an MPI_Comm object (MPI intra-communicator). The PET local process id within each virtual machine is equal to the MPI_Comm_rank in the local MPI_Comm context whereas the PET process id is equal to the MPI_Comm_rank in MPI_COMM_WORLD. The PET process id is used within the VM methods to determine the virtual memory space a PET is operating in.
In order to provide a migration path for legacy MPI-applications the VM offers accessor functions to its MPI_Comm object. Once obtained this object may be used in explicit user-code MPI calls within the same context.

41.5 Class API

41.5.1 ESMF_VMAllFullReduce - AllFullReduce 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMAllFullReduce()
subroutine ESMF_VMAllFullReduceI4(vm, sendData, recvData, count, &
  reduceflag,  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),             intent(in)              :: vm
  integer(ESMF_KIND_I4),     intent(in)              :: sendData(:)
  integer(ESMF_KIND_I4),     intent(out)             :: recvData
```

```
    integer,                  intent(in)              :: count
    type(ESMF_ReduceFlag),    intent(in)              :: reduceflag
    type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
    type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
    integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that performs an AllFullReduce on contigous data of kind `ESMF_KIND_I4` across the `ESMF_VM` object performing the specified operation.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.

`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.2 ESMF_VMAllFullReduce - AllFullReduce 4-byte reals

INTERFACE:

```
  ! Private name; call using ESMF_VMAllFullReduce()
  subroutine ESMF_VMAllFullReduceR4(vm, sendData, recvData, count, &
    reduceflag, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
    type(ESMF_VM),            intent(in)              :: vm
    real(ESMF_KIND_R4),       intent(in)              :: sendData(:)
    real(ESMF_KIND_R4),       intent(out)             :: recvData
    integer,                  intent(in)              :: count
    type(ESMF_ReduceFlag),    intent(in)              :: reduceflag
    type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
    type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
    integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that performs an AllFullReduce on contigous data of kind `ESMF_KIND_R4` across the `ESMF_VM` object performing the specified operation.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.3 ESMF_VMAllFullReduce - AllFullReduce 8-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMAllFullReduce()
subroutine ESMF_VMAllFullReduceR8(vm, sendData, recvData, count, &
  reduceflag, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)               :: vm
real(ESMF_KIND_R8),       intent(in)               :: sendData(:)
real(ESMF_KIND_R8),       intent(out)              :: recvData
integer,                  intent(in)               :: count
type(ESMF_ReduceFlag),    intent(in)               :: reduceflag
type(ESMF_BlockingFlag),  intent(in),   optional   :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional   :: commhandle
integer,                  intent(out),  optional   :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that performs an AllFullReduce on contigous data of kind ESMF_KIND_R4 across the ESMF_VM object performing the specified operation.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.4 ESMF_VMAllReduce - AllReduce 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMAllReduce()
subroutine ESMF_VMAllReduceI4(vm, sendData, recvData, count, reduceflag, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)                :: vm
integer(ESMF_KIND_I4),    intent(in)                :: sendData(:)
integer(ESMF_KIND_I4),    intent(out)               :: recvData(:)
integer,                  intent(in)                :: count
type(ESMF_ReduceFlag),    intent(in)                :: reduceflag
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that performs an AllReduce on contigous data of kind ESMF_KIND_I4 across the ESMF_VM object performing the specified operation.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.5 ESMF_VMAllReduce - AllReduce 4-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMAllReduce()
subroutine ESMF_VMAllReduceR4(vm, sendData, recvData, count, reduceflag, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
real(ESMF_KIND_R4),       intent(in)              :: sendData(:)
real(ESMF_KIND_R4),       intent(out)             :: recvData(:)
integer,                  intent(in)              :: count
type(ESMF_ReduceFlag),    intent(in)              :: reduceflag
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that performs an AllReduce on contigous data of kind `ESMF_KIND_R4` across the `ESMF_VM` object performing the specified operation.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.

`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.6 ESMF_VMAllReduce - AllReduce 8-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMAllReduce()
subroutine ESMF_VMAllReduceR8(vm, sendData, recvData, count, reduceflag, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),           intent(in)              :: vm
real(ESMF_KIND_R8),      intent(in)              :: sendData(:)
real(ESMF_KIND_R8),      intent(out)             :: recvData(:)
integer,                 intent(in)              :: count
type(ESMF_ReduceFlag),   intent(in)              :: reduceflag
type(ESMF_BlockingFlag), intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),   intent(out),  optional  :: commhandle
integer,                 intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that performs an AllReduce on contigous data of kind `ESMF_KIND_R8` across the `ESMF_VM` object performing the specified operation.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements in sendData on each of the PETs.

reduceflag Reduction operation.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

- `ESMF_BLOCKING` Block until local operation has completed.
- `ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.7 ESMF_VMBarrier - VM wide barrier

INTERFACE:

```
subroutine ESMF_VMBarrier(vm, rc)
```

ARGUMENTS:

```
type(ESMF_VM),  intent(in)              :: vm
integer,        intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that blocks calling PET until all of the PETs have issued the call.
The arguments are:

vm `ESMF_VM` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.8 ESMF_VMGather - Gather 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMGather()
subroutine ESMF_VMGatherI4(vm, sendData, recvData, count, root, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),             intent(in)               :: vm
integer(ESMF_KIND_I4),     intent(in)               :: sendData(:)
integer(ESMF_KIND_I4),     intent(out)              :: recvData(:)
integer,                   intent(in)               :: count
integer,                   intent(in)               :: root
type(ESMF_BlockingFlag),   intent(in),   optional   :: blockingflag
type(ESMF_CommHandle),     intent(out),  optional   :: commhandle
integer,                   intent(out),  optional   :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that gathers contigous data of kind ESMF_KIND_I4 from the PETs of an ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. Only the recvData array specified by the root PET will be used by this method.

count Number of elements to be send from root to each of the PETs.

root Id of the root PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.9 ESMF_VMGather - Gather 4-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMGather()
subroutine ESMF_VMGatherR4(vm, sendData, recvData, count, root, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
    type(ESMF_VM),           intent(in)              :: vm
    real(ESMF_KIND_R4),      intent(in)              :: sendData(:)
    real(ESMF_KIND_R4),      intent(out)             :: recvData(:)
    integer,                 intent(in)              :: count
    integer,                 intent(in)              :: root
    type(ESMF_BlockingFlag), intent(in),   optional  :: blockingflag
    type(ESMF_CommHandle),   intent(out),  optional  :: commhandle
    integer,                 intent(out),  optional  :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that gathers contigous data of kind ESMF_KIND_R4 from the PETs of an ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. Only the recvData array specified by the root PET will be used by this method.

count Number of elements to be send from root to each of the PETs.

root Id of the root PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.10 ESMF_VMGather - Gather 8-byte reals

INTERFACE:

```
  ! Private name; call using ESMF_VMGather()
  subroutine ESMF_VMGatherR8(vm, sendData, recvData, count, root, &
    blockingflag, commhandle, rc)
```

ARGUMENTS:

```
    type(ESMF_VM),           intent(in)              :: vm
    real(ESMF_KIND_R8),      intent(in)              :: sendData(:)
    real(ESMF_KIND_R8),      intent(out)             :: recvData(:)
    integer,                 intent(in)              :: count
    integer,                 intent(in)              :: root
    type(ESMF_BlockingFlag), intent(in),   optional  :: blockingflag
    type(ESMF_CommHandle),   intent(out),  optional  :: commhandle
    integer,                 intent(out),  optional  :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that gathers contigous data of kind ESMF_KIND_R8 from the PETs of an ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. All PETs must specify a valid source array.

recvData Contigous data array for data to be received. Only the recvData array specified by the root PET will be used by this method.

count Number of elements to be send from root to each of the PETs.

root Id of the root PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.11 ESMF_VMGet - Get VM internals

INTERFACE:

```
subroutine ESMF_VMGet(vm, localPet, petCount, peCount, mpiCommunicator, &
  okOpenMpFlag, rc)
```

ARGUMENTS:

```
type(ESMF_VM),       intent(in)                :: vm
integer,             intent(out),   optional   :: localPet
integer,             intent(out),   optional   :: petCount
integer,             intent(out),   optional   :: peCount
integer,             intent(out),   optional   :: mpiCommunicator
type(ESMF_Logical),  intent(out),   optional   :: okOpenMpFlag
integer,             intent(out),   optional   :: rc
```

DESCRIPTION:

Get internal information about the specified ESMF_VM object.
The arguments are:

vm Queried ESMF_VM object.

[localPet] Upon return this holds the id of the PET that instantiates the local user code.

[petCount] Upon return this holds the number of PETs in the specified ESMF_VM object.

[peCount] Upon return this holds the number of PEs referenced by the specified ESMF_VM object.

[mpiCommunicator] Upon return this holds the MPI intra-communicator used by the specified ESMF_VM object. This communicator may be used for user-level MPI communications. It is recommended that the user duplicates the communicator via MPI_Comm_Dup() in order to prevent any interference with ESMF communications.

[okOpenMpFlag] Upon return this holds a flag indicating whether user-level OpenMP threading is supported by the specified ESMF_VM object.

ESMF_TRUE User-level OpenMP threading is supported.

ESMF_FALSE User-level OpenMP threading is not supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.12 ESMF_VMGetGlobal - Get Global VM

INTERFACE:

```
subroutine ESMF_VMGetGlobal(vm, rc)
```

ARGUMENTS:

```
  type(ESMF_VM), intent(out)             :: vm
  integer,       intent(out), optional   :: rc
```

DESCRIPTION:

Get the global default ESMF_VM object. This is the ESMF_VM object that was created during ESMF_Initialize() and is the ultimate parent of all ESMF_VM objects in an ESMF application.
The arguments are:

vm Upon return this holds the global default ESMF_VM object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.13 ESMF_VMGetPETLocalInfo - Get VM PET local internals

INTERFACE:

```
subroutine ESMF_VMGetPETLocalInfo(vm, pet, peCount, ssiId, threadCount, &
  threadId, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),  intent(in)                 :: vm
  integer,        intent(in)                 :: pet
  integer,        intent(out),  optional     :: peCount
  integer,        intent(out),  optional     :: ssiId
  integer,        intent(out),  optional     :: threadCount
  integer,        intent(out),  optional     :: threadId
  integer,        intent(out),  optional     :: rc
```

DESCRIPTION:

Get internal information about the specified PET within the specified ESMF_VM object.
The arguments are:

vm Queried ESMF_VM object.

pet Queried PET id within the specified ESMF_VM object.

[peCount] Upon return this holds the number of PEs associated with the specified PET in the ESMF_VM object.

[ssiId] Upon return this holds the id of the single-system image (SSI) the specified PET is running on.

[threadCount] Upon return this holds the number of PETs in the specified PET's thread group.

[threadId] Upon return this holds the thread id of the specified PET within the ESMF_VM object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.14 ESMF_VMPrint - Print VM internals

INTERFACE:

```
subroutine ESMF_VMPrint(vm, rc)
```

ARGUMENTS:

```
type(ESMF_VM),  intent(in)              :: vm
integer,        intent(out),  optional  :: rc
```

DESCRIPTION:

Prints internal information about the specified ESMF_VM to stdout.
The arguments are:

vm Specified ESMF_VM object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.15 ESMF_VMRecv - Receive 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMRecv()
subroutine ESMF_VMRecvI4(vm, recvData, count, src, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
integer(ESMF_KIND_I4),    intent(in)              :: recvData(:)
integer,                  intent(in)              :: count
integer,                  intent(in)              :: src
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Receive contigous data of kind ESMF_KIND_I4 from a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

recvData Contigous data array for data to be received.

count Number of elements to be received.

src Id of the source PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.16 ESMF_VMRecv - Receive 4-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMRecv()
subroutine ESMF_VMRecvR4(vm, recvData, count, src, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
real(ESMF_KIND_R4),       intent(in)              :: recvData(:)
integer,                  intent(in)              :: count
integer,                  intent(in)              :: src
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Receive contigous data of kind ESMF_KIND_R4 from a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

recvData Contigous data array for data to be received.

count Number of elements to be received.

src Id of the source PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.17 ESMF_VMRecv - Receive 8-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMRecv()
subroutine ESMF_VMRecvR8(vm, recvData, count, src, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)               :: vm
real(ESMF_KIND_R8),       intent(in)               :: recvData(:)
integer,                  intent(in)               :: count
integer,                  intent(in)               :: src
type(ESMF_BlockingFlag),  intent(in),   optional   :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional   :: commhandle
integer,                  intent(out),  optional   :: rc
```

DESCRIPTION:

Receive contigous data of kind ESMF_KIND_R8 from a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

recvData Contigous data array for data to be received.

count Number of elements to be received.

src Id of the source PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.18 ESMF_VMScatter - Scatter 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMScatter()
subroutine ESMF_VMScatterI4(vm, sendData, recvData, count, root, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)               :: vm
integer(ESMF_KIND_I4),    intent(in)               :: sendData(:)
integer(ESMF_KIND_I4),    intent(out)              :: recvData(:)
integer,                  intent(in)               :: count
integer,                  intent(in)               :: root
type(ESMF_BlockingFlag),  intent(in),   optional   :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional   :: commhandle
integer,                  intent(out),  optional   :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that scatters contigous data of kind ESMF_KIND_I4 across the PETs of an ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. Only the sendData array specified by the root PET will be used by this method.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements to be send from root to each of the PETs.

root Id of the root PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.19 ESMF_VMScatter - Scatter 4-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMScatter()
subroutine ESMF_VMScatterR4(vm, sendData, recvData, count, root, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),            intent(in)               :: vm
  real(ESMF_KIND_R4),       intent(in)               :: sendData(:)
  real(ESMF_KIND_R4),       intent(out)              :: recvData(:)
  integer,                  intent(in)               :: count
  integer,                  intent(in)               :: root
  type(ESMF_BlockingFlag),  intent(in),   optional   :: blockingflag
  type(ESMF_CommHandle),    intent(out),  optional   :: commhandle
  integer,                  intent(out),  optional   :: rc
```

DESCRIPTION:

Collective ESMF_VM communication call that scatters contigous data of kind ESMF_KIND_R4 across the PETs of an ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data arry holding data to be send. Only the sendData array specified by the root PET will be used by this method.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements to be send from `root` to each of the PETs.

root Id of the `root` PET within the `ESMF_VM` object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.
`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.20 ESMF_VMScatter - Scatter 8-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMScatter()
subroutine ESMF_VMScatterR8(vm, sendData, recvData, count, root, &
  blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
real(ESMF_KIND_R8),       intent(in)              :: sendData(:)
real(ESMF_KIND_R8),       intent(out)             :: recvData(:)
integer,                  intent(in)              :: count
integer,                  intent(in)              :: root
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Collective `ESMF_VM` communication call that scatters contigous data of kind `ESMF_KIND_R8` across the PETs of an `ESMF_VM` object.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data arry holding data to be send. Only the `sendData` array specified by the `root` PET will be used by this method.

recvData Contigous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements to be send from `root` to each of the PETs.

root Id of the `root` PET within the `ESMF_VM` object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.
`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.21 ESMF_VMSend - Send 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMSend()
subroutine ESMF_VMSendI4(vm, sendData, count, dst, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),            intent(in)                :: vm
  integer(ESMF_KIND_I4),    intent(in)                :: sendData(:)
  integer,                  intent(in)                :: count
  integer,                  intent(in)                :: dst
  type(ESMF_BlockingFlag),  intent(in),   optional    :: blockingflag
  type(ESMF_CommHandle),    intent(out),  optional    :: commhandle
  integer,                  intent(out),  optional    :: rc
```

DESCRIPTION:

Send contigous data of kind `ESMF_KIND_I4` to a PET within the same `ESMF_VM` object.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data array holding data to be send.

count Number of elements to be send.

dst Id of the destination PET within the `ESMF_VM` object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.

`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.22 ESMF_VMSend - Send 4-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMSend()
subroutine ESMF_VMSendR4(vm, sendData, count, dst, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),            intent(in)                :: vm
  real(ESMF_KIND_R4),       intent(in)                :: sendData(:)
  integer,                  intent(in)                :: count
  integer,                  intent(in)                :: dst
  type(ESMF_BlockingFlag),  intent(in),   optional    :: blockingflag
  type(ESMF_CommHandle),    intent(out),  optional    :: commhandle
  integer,                  intent(out),  optional    :: rc
```

DESCRIPTION:

Send contigous data of kind ESMF_KIND_R4 to a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data array holding data to be send.

count Number of elements to be send.

dst Id of the destination PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.
ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.23 ESMF_VMSend - Send 8-byte reals

INTERFACE:

```
! Private name; call using ESMF_VMSend()
subroutine ESMF_VMSendR8(vm, sendData, count, dst, blockingflag, &
  commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),            intent(in)              :: vm
  real(ESMF_KIND_R8),       intent(in)              :: sendData(:)
  integer,                  intent(in)              :: count
  integer,                  intent(in)              :: dst
  type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
  type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
  integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Send contigous data of kind ESMF_KIND_R8 to a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data array holding data to be send.

count Number of elements to be send.

dst Id of the destination PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.
ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.24 ESMF_VMSendRecv - SendRecv 4-byte integers

INTERFACE:

```
! Private name; call using ESMF_VMSendRecv()
subroutine ESMF_VMSendRecvI4(vm, sendData, sendCount, dst, &
  recvData, recvCount, src, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
  type(ESMF_VM),            intent(in)                :: vm
  integer(ESMF_KIND_I4),    intent(in)                :: sendData(:)
  integer,                  intent(in)                :: sendCount
  integer,                  intent(in)                :: dst
  integer(ESMF_KIND_I4),    intent(in)                :: recvData(:)
  integer,                  intent(in)                :: recvCount
  integer,                  intent(in)                :: src
  type(ESMF_BlockingFlag),  intent(in),   optional    :: blockingflag
  type(ESMF_CommHandle),    intent(out),  optional    :: commhandle
  integer,                  intent(out),  optional    :: rc
```

DESCRIPTION:

Send contigous data of kind `ESMF_KIND_I4` to a PET within the same `ESMF_VM` object while receiving contigous data of kind `ESMF_KIND_I4` from a PET within the same `ESMF_VM` object.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data array holding data to be send.

sendCount Number of elements to be send.

dst Id of the destination PET within the `ESMF_VM` object.

recvData Contigous data array for data to be received.

recvCount Number of elements to be received.

src Id of the source PET within the `ESMF_VM` object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

`ESMF_BLOCKING` Block until local operation has completed.

`ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.25 ESMF_VMSendRecv - SendRecv 4-byte real

INTERFACE:

```
! Private name; call using ESMF_VMSendRecv()
subroutine ESMF_VMSendRecvR4(vm, sendData, sendCount, dst, &
  recvData, recvCount, src, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
real(ESMF_KIND_R4),       intent(in)              :: sendData(:)
integer,                  intent(in)              :: sendCount
integer,                  intent(in)              :: dst
real(ESMF_KIND_R4),       intent(in)              :: recvData(:)
integer,                  intent(in)              :: recvCount
integer,                  intent(in)              :: src
type(ESMF_BlockingFlag),  intent(in),   optional  :: blockingflag
type(ESMF_CommHandle),    intent(out),  optional  :: commhandle
integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

Send contigous data of kind `ESMF_KIND_R4` to a PET within the same `ESMF_VM` object while receiving contigous data of kind `ESMF_KIND_R4` from a PET within the same `ESMF_VM` object.
The arguments are:

vm `ESMF_VM` object.

sendData Contigous data array holding data to be send.

sendCount Number of elements to be send.

dst Id of the destination PET within the `ESMF_VM` object.

recvData Contigous data array for data to be received.

recvCount Number of elements to be received.

src Id of the source PET within the `ESMF_VM` object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

- `ESMF_BLOCKING` Block until local operation has completed.
- `ESMF_NONBLOCKING` Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument `blockingflag`).

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

41.5.26 ESMF_VMSendRecv - SendRecv 8-byte real

INTERFACE:

```
! Private name; call using ESMF_VMSendRecv()
subroutine ESMF_VMSendRecvR8(vm, sendData, sendCount, dst, &
  recvData, recvCount, src, blockingflag, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),            intent(in)              :: vm
real(ESMF_KIND_R8),       intent(in)              :: sendData(:)
integer,                  intent(in)              :: sendCount
integer,                  intent(in)              :: dst
real(ESMF_KIND_R8),       intent(in)              :: recvData(:)
```

```
integer,                   intent(in)                :: recvCount
integer,                   intent(in)                :: src
type(ESMF_BlockingFlag),   intent(in),   optional    :: blockingflag
type(ESMF_CommHandle),     intent(out),  optional    :: commhandle
integer,                   intent(out),  optional    :: rc
```

DESCRIPTION:

Send contigous data of kind ESMF_KIND_R8 to a PET within the same ESMF_VM object while receiving contigous data of kind ESMF_KIND_R8 from a PET within the same ESMF_VM object.
The arguments are:

vm ESMF_VM object.

sendData Contigous data array holding data to be send.

sendCount Number of elements to be send.

dst Id of the destination PET within the ESMF_VM object.

recvData Contigous data array for data to be received.

recvCount Number of elements to be received.

src Id of the source PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING Block until local operation has completed.

ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] A communication handle will be returned in case of a non-blocking request (see argument blockingflag).

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.27 ESMF_VMThreadBarrier - PET thread group wide barrier

INTERFACE:

```
subroutine ESMF_VMThreadBarrier(vm, rc)
```

ARGUMENTS:

```
type(ESMF_VM),  intent(in)               :: vm
integer,        intent(out),  optional   :: rc
```

DESCRIPTION:

Partially collective ESMF_VM communication call that blocks calling PET until all of the PETs that are running under the same POSIX process have issued the call.
The arguments are:

vm ESMF_VM object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

41.5.28 ESMF_VMWait - Wait for non-blocking VM communication to complete

INTERFACE:

```
subroutine ESMF_VMWait(vm, commhandle, rc)
```

ARGUMENTS:

```
type(ESMF_VM),           intent(in)                :: vm
type(ESMF_CommHandle),   intent(in)                :: commhandle
integer,                 intent(out),  optional    :: rc
```

DESCRIPTION:

Wait for non-blocking VM communication to complete.
The arguments are:

vm `ESMF_VM` object.

commhandle Handle specifying a previous non-blocking communication request.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

42 Bibliography

References

[1] Some notes on the iso8601 time and date specification standard. http://www.mcs.vuw.ac.nz/technical/software/SGML/doc/iso8601/ISO8601.html.

[2] P.W. Jones. First- and second-order conservative remapping schemes for grids in spherical coordinates. *Monthly Weath. Rev.*, 127:2204–2210, 1999.

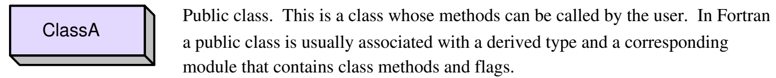
[3] Rumbaugh, J., I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[4] Seidelman, P.K. *Explanatory Supplement to the Astronomical Almanac*. University Science Books, 1992.

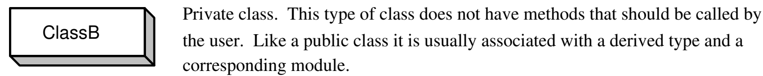
Part V
Appendices

43 Appendix A: A Brief Introduction to UML

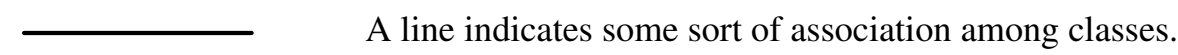
The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [3].



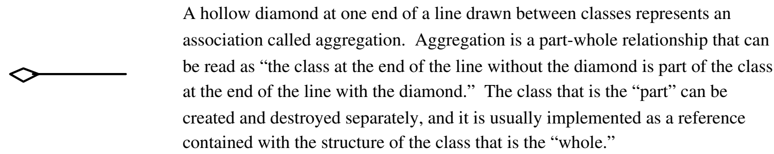
Public class. This is a class whose methods can be called by the user. In Fortran a public class is usually associated with a derived type and a corresponding module that contains class methods and flags.



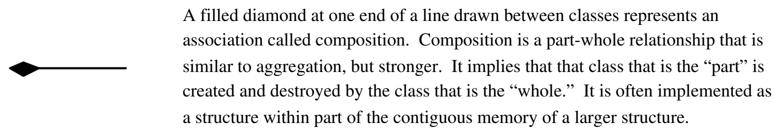
Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.



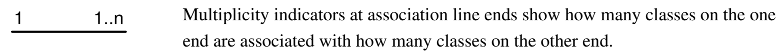
A line indicates some sort of association among classes.



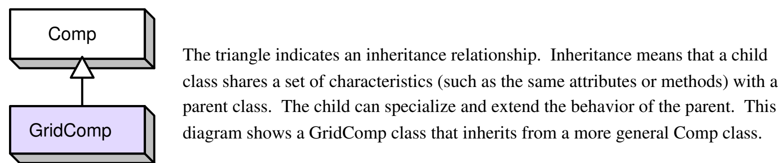
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as "the class at the end of the line without the diamond is part of the class at the end of the line with the diamond." The class that is the "part" can be created and destroyed separately, and it is usually implemented as a reference contained with the structure of the class that is the "whole."



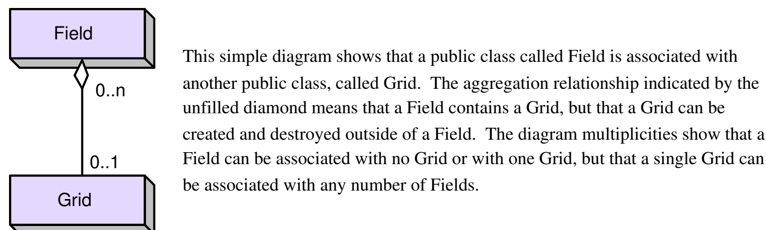
A filled diamond at one end of a line drawn between classes represents an association called composition. Composition is a part-whole relationship that is similar to aggregation, but stronger. It implies that that class that is the "part" is created and destroyed by the class that is the "whole." It is often implemented as a structure within part of the contiguous memory of a larger structure.



Multiplicity indicators at association line ends show how many classes on the one end are associated with how many classes on the other end.



The triangle indicates an inheritance relationship. Inheritance means that a child class shares a set of characteristics (such as the same attributes or methods) with a parent class. The child can specialize and extend the behavior of the parent. This diagram shows a GridComp class that inherits from a more general Comp class.



This simple diagram shows that a public class called Field is associated with another public class, called Grid. The aggregation relationship indicated by the unfilled diamond means that a Field contains a Grid, but that a Grid can be created and destroyed outside of a Field. The diagram multiplicities show that a Field can be associated with no Grid or with one Grid, but that a single Grid can be associated with any number of Fields.